第 2 章　Object Pascal 语言

Delphi 的编程语言是以 Pascal 为基础的。Pascal 语言具有可读性好、编写容易的特点，这使得它很适合作为基础的程序开发语言。同时，使用编译器创建的应用程序只生成单个可执行文件（.EXE），正是这种结合，使得 Pascal 成为 Delphi 这种先进开发环境的编程语言。

本章中，我们把 Delphi 的可视化元素放在一边，将主要讨论 Object Pascal 的主要特点，并讲解如何在事件处理过程和其它应用程序中，使用它来编制程序代码。首先，本章将讲解 Delphi 应用程序中最常用的 Object Pascal 语法和面向对象技术；然后介绍一些 Object Pascal 语言的高级技术。如果读者完全不熟悉 Pascal 编程，可以参阅一些基础的 Pascal 教程。如果具有一定编程经验，并能熟练地使用其它流行的程序语言，就将会在本章的 Object Pascal 中发现一些相同的概念。如果读者已经熟悉了 Borland Pascal 和面向对象技术，就可以快速浏览或直接跳过本章。

2.1　Object Pascal 语言基础

2.1.1　Object Pascal 入门

一般来说，使用 Delphi 开发的程序有以下三种：

1. Windows 图形界面程序

Windows 图形界面程序包括了目前 Windows 平台上绝大多数应用程序，这些程序通过窗体和对话框与用户进行信息交互，实现一定的功能。如 Microsoft 公司的 Office 系列软件和正在使用的 Delphi 6.0 等。

2. 控制台程序

控制台程序是指一些没有图形用户界面的 32 位 Windows 应用程序。通常是在类似以前的 DOS 环境下运行。这些程序很少要求用户输入大量的信息，一般只实现特定的功能。控制台程序的代码较小，占用的系统资源少。编译、链接的速度比较快。在本章中将主要采用控制台程序介绍 Object Pascal 语言。

3. 服务器程序

服务器程序可以接受客户应用程序的请求，处理这些请求，并将结果信息返回客户应用程序。服务器应用程序一般在后台运行，也不需要太多的信息交互。

下面我们通过一个具体的控制台程序的例子，使读者对在 Delphi 中生成 Object Pascal 程序，以及 Object Pascal 程序的一般结构有一个比较直观的认识。

在 Delphi 集成开发环境中，激活菜单 File/New 打开 New Items 对话框。选中 New 标签页中的 Console Application 选项，按下 OK 按钮。系统会自动创建下面的控制台程序工程，并在代码编辑窗口中自动打开 Project1.dpr 文件，修改 Project1.dpr 文件如下，注意程序中的

黑体部分：

```
program Project1;
{$APPTYPE CONSOLE}
//uses SysUtils;
var
  str:string;

begin
  // Insert user code here
  writeln('您好,这是一个示范程序，请输入一行文字: ');
  readln(str);
  writeln('您输入的是: ',str);

  readln;
end.
```

激活菜单项 File/Save All 将工程文件保存至目录 D:\Delphi\Samples\Ex2_1 中。黑体部分为添加代码。本章后面的例子都可以在该工程的基础上来方便的实现。

按 F9 键编译、链接、运行程序。在程序提示：“您好, 这是一个示范程序，请输入一行文字：”后，输入“早上好！”。程序将输出结果：“您输入的是：早上好！”。

下面是对该程序的几点说明：

- 程序第二行中{$APPTYPE CONSOLE}是一个编译器指令，它告诉编译器这个程序是一个控制台程序。
- uses SysUtils 语句前加了两个斜线，将这条语句改成注释语句屏蔽掉了（注释语句将在下面介绍），因为 SysUtils 单元在这个例子中用不到。
- writeln 和 readln 两个函数分别输出和输入一行字符，后面将详细介绍两个函数。程序最后的 readln 语句是为了锁定窗口，否则输出结果将很快消失。

我们通过这个简单的例子，作为读者对 Object Pascal 的入门。下面将详细介绍 Object Pascal 的语法。

2.1.2 注释语句

作为起点，我们首先介绍在 Object Pascal 代码中怎样写注释。Object Pascal 支持三种类型的注释：

- 花括号注释：组合符号“{”和“}”的成对使用表示它们之间的内容是注释部分；
- 圆括号/星号注释：组合符号“（*”和“*）”的成对使用表示它们之间的内容是注释部分；
- C++风格的双斜杠注释：符号“//”的单独使用表示后面的内容是注释部分；

我们看下面的例子：

```
{花括号注释}
( *圆括号/星号注释* )
// C++风格的注释
```

前两种注释在本质上是相同的，编译器把处于限定符头和限定符尾中间的内容当作注释。花括号和圆括号/星号比较适合在大段注释时使用。如果在“{”或“(*”后面是一个“$”符号时，表示该句为一个编译器指令，与普通的注释不同，通常用来对编译过程进行设置。如上例中的第二句{$APPTYPE CONSOLE}。

对于 C++风格的注释来说，双斜杠后面到行尾的内容被认为是注释。比较适用于单行和少量几行注释的情况。

注意，相同类型的注释不能嵌套使用。虽然不同类型的注释进行嵌套在语法上是合法的，但我们不建议这样做。这里是一些例子：

```
{ ( *这是合法的* ) }
( * {这是合法的} * )
( * ( *这是非法的* ) * )
{ {这是非法的} }
```

2.1.3 标识符（Identifier）

Object Pascal 语言使用的标识符包括字母 A～Z，a～z，数字 0～9 以及其它一些标准字符。

下面的单个字符作为特殊符号存在：

$	&	*	#	‘	(	)	[	]
{	}	^	;	:	@	<	=	>
,	.	?	+	/				

下面的字符组合作为单个的特殊符号存在：

（*	*）	(.	.)	..	//
:=	<>	>=	<=		

注意：

[ ]与（. .）分别对应，{ }与(* *)分别对应。含义完全相同，可以相互替代。

在 Object Pascal 中，标识符用来标志变量，常量，属性，类，对象，过程，函数，程序，组件库等。标识符可以由任意长的不带空格的字符串组成，但对编译器只有前面 255 个字符有效。其中，标识符的第一个字符必须是字母和下划线，其余字符可以是字母，数字或下划线。通常，标识符由一个或多个具有适当意义的英文单词组成。

Pascal 语言对字母的大小写是不敏感的。在编程过程中，最好每个单词的首字母大写，其它字母小写，以便于区分。

2.1.4 保留字（Reserved Word）和指令字（Directive）

Object Pascal 语言定义了 67 个保留字，不能被定义为标识符，具体如下：

and	array	as	asm	begin	case
class	const	constructor	destructor	interface	div
do	down	to	else	end	except
exports	file	finalization	finally	for	function
goto	if	implementation	in	inherite	

initialization	inline	is	interface	label	library
mod	nil	not	object	of	or
out	packed	procedure	program	property	raise
record	repeat	resource	string	set	shl
shr	then	thread	var	to	try
type	unit	until	uses	var	while
with	xor				

Object Pascal 还定义了 39 个指令字，它们具有特殊含义，但是，当用户重新定义了指令字后，在作用域内它们就失去了原来的意义。具体如下：

absolute	abstract	assembler	automated
cdecl	contains	default	dispid
dynamic	export	external	far
forward	implements	index	message
name	near	nodefault	overload
override	package	pascal	private
protected	public	published	read
readonly	register	reintroduce	requires
resident	safecall	stdcall	stored
virtual	write	writeonly	

其中，private，protected，public，published，automated，on，at 是指令字，但应当视其为保留字，不要定义和它们同名的标识符。

在使用过程中，不用担心因为不小心而错误的将保留字或指令字定义为标识符。在 Delphi 集成开发环境的代码编辑器中，会以黑体显示保留字和指令字。这样就大大方便了用户的使用。

2.1.5 数据类型

Object Pascal 的最大特点是，它的数据类型特别严谨，这表示传递给过程或函数的实参必须和定义过程或函数时的形参的类型相同。我们不会在 Pascal 中看到一些著名编译器例如 C 编译器所提示的可疑的指针转换等编译警告信息。这是因为 Object Pascal 编译器不允许用一种类型的指针去调用形参为另一种类型的函数（无类型的指针除外）。

Object Pascal 语言提供的数据类型非常丰富。有简单数据类型（Simple），字符串数据类型（String），结构数据类型（Struct），指针数据类型（pointer），函数和过程数据类型（procedural），变体数据类型（variant）等。下面我们针对各种数据类型一一加以介绍。

1. 简单数据类型（Simple）

简单数据类型包括有序数据类型（ordinal）和实数数据类型（real）。其中，有序数据类型又包括整数类型，字符类型，布尔类型，枚举类型和子界类型。它们的关系如图 2-1 所示。

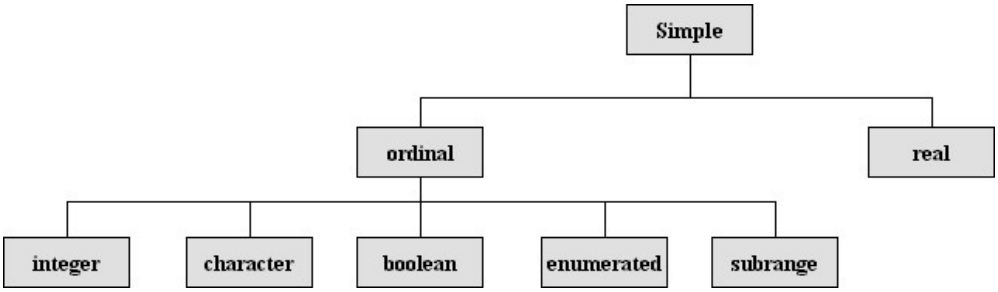

图 2-1 简单数据类型

（1）整数类型（integer）

整数类型如下表 2-1 中的各种类型。

表 2-1 Object Pascal 语言中的整数类型

整数类型	数值范围
Integer	signed 32 bit
Cardinal	unsigned 32 bit
Shortint	signed 8 bit
Smallint	signed 16 bit
Longint	signed 32 bit
int64	signed 64 bit
Byte	unsigned 8 bit
Word	unsigned 16 bit
Longword	unsigned 32 bit

（2）字符类型

Delphi 可以使用三种类型的字符变量：

- AnsiChar：标准的 1 字节的 ANSI 字符。
- WideChar：2 字节的 Unicode 字符。Unicode 字符集的前 256 个字符与 ANSI 字符集相同。
- Char：相当于 WideChar。

需要注意的是，因为一个字符在长度上并不表示一个字节，所以不能在应用程序中对字符长度进行硬编码，而应该使用 Sizeof() 函数。注意 Sizeof() 标准函数返回类型或实例的字节长度。

（3）布尔类型

布尔类型（boolean）包括四种：Boolean，ByteBool，WordBool，LongBool。其中 Boolean 和 ByteBool 为单字节，WordBool 为双字节，LongBool 为四字节。Object Pascal 预定义了了两个常量标示符 false 和 true。可以将、Boolean 类型的数据赋值为 false 和 true，对应的序数值为 0 和 1。ByteBool，WordBool，LongBool 类型对应的数据为 0 时，可以认为是 false；不为 0 时，可以认为是 true。

（4）枚举类型

枚举类型（enumerated）是由一组有序的标识符组成的，说明列出了所有这种类型可以包括的值，如下面的例子。

```
type
  Tdays=( Sunday ,Monday,Tuesday,Wednesday,Thursday,Friday,Saturday);
```

可以定义上述枚举类型的变量：

```
var
    DayOfWeek:TDays;
```

在枚举型中，括号里的每一个值都有一个由说明它的位置决定的整形值。例如 Sunday 有整形值 0，Monday 有整形值 1 等。用户可以把 DayOfWeek 说明为一个整形变量，并将一星期的每一天赋一个整型值以达到相同的效果，但用枚举型会使得程序可读性好，编写容易。当在枚举型中列出值时，同时说明了这个值是一个标识符。例如程序中如果已经含有 TDays 类型且说明了 DayOfWeeks 变量，则程序中便不能再使用 Monday 变量，因为它已经被说明为标识符了。

（5）子界类型

子界类型（subrange）是下列这些类型中某范围内的值：整型、布尔型、字符型或枚举型。在用户想限制一个变量的取值范围时，子界型是非常有用的。

```
type
  Thours = 0..23;
  TValidLetter = 'A' .. 'F';
  TDays = ( Sunday ,Monday,Tuesday,Wednesday,Thursday,
        Friday,Saturday);    {枚举型}
  TWorkDay = Monday..Friday; {一个TDays型的子界}
```

子界型限定了变量的可能取值范围。当范围检查打开时，（在库单元的 Implementation 后面有{$R*.DFM}字样表示范围检查打开，否则您可以在 Options/Project/Complier Options 中选择 Range Cheking 来打开范围检查），如果变量取到子界以外的值，会出现一个范围检查错误。

（6）实数数据

Object Pascal 语言中的实数数据（real）类型如表 2-2 所示。

表 2-2 Object Pascal 语言中的实数数据类型

实数类型	范围	有效位数	字节数
Real48	2．9*10^-9..1.7*10^38	11	6
Single	1.5*10^-45..3.4*10^38	7	4
Double	5*10^-324..1.7*10^308	15	8
Extended	3.6*10^-4951..1.1*10^4932	19	10
Comp	-2^63+1..2^63-1	19	8
Currency	-922337203685477.5808.. 922337203685477.5807	19	8

2. 字符串类型（String）

字符串是代表一组字符的变量类型，每一种语言都有自己的字符串类型的存储和使用方法。在 Object Pascal 中，通常用一对单引号来把字符串括起来，例如'A String'。

Pascal 类型有下列几种不同的字符串类型来满足程序的要求：

- AnsiString 这是 Pascal 缺省的字符串类型，它由 AnsiChar 字符组成，其长度没有限制，同时与 null 结束的字符串相兼容。
- ShortString 保留该类型是为了向后兼容 Delphi，它的长度限制在 255 个字符内。
- WideString 功能上类似于 AnsiString，但它是由 WideChar 字符组成的。
- PChar 指向 null 结束的 Char 字符串的指针，类似于 C 语言的 char * 或 lpstr 类型。
- PAnsiChar 指向 null 结束的 AnsiChar 字符串的指针。
- PWideChar 指向 null 结束的 WideChar 字符串的指针。

缺省情况下，如果用如下的代码来定义字符串，编译器认为是 AnsiString 字符串：

```
var
  Str:string; //编译器认为Str的类型是AnsiString
```

可以用编译开关$H 来将 string 类型定义为 ShortString，当$H 编译开关的值为负时，string 变量是 ShortString 类型；当$ H 编译开关的值为正时（缺省情况），字符串变量是 AnsiString 类型。如下面的程序：

```
var
  {$H-}
  E1:string; //E1是ShortString类型
  {$H+}
  S2:string; //S2是AnsiString类型
```

使用$H 规则的一个例外是，如果在定义时特地指定了长度（最大在 255 个字符内），那么就总是 ShortString，例如：

```
var
  S: string[63]; //63个字符的ShortString字符串
```

（1）AnsiString 类型

AnsiString (或长字符串)类型是在 Delphi 2.0 开始引入的，因为 Delphi 1.0 的用户特别需要一个容易使用而且没有 255 个字符限制的字符串类型，而 AnsiString 正好能满足这些要求。虽然 AnsiString 在外表上跟以前的字符串类型几乎相同，但它是动态分配的并有自动回收功能，正是因为这个功能 AnsiString 有时被称为生存期自管理类型。Object Pascal 能根据需要为字符串分配空间，所以不用像在 C / C++中所担心的为中间结果分配缓冲区。另外，AnsiString 字符串总是以 null 字符结束的，这使得 AnsiString 字符串能与 Win32 API 中的字符串兼容。实际上，AnsiString 类型是一个指向在堆栈中的字符串结构的指针。

（2）ShortString 类型

ShortString 类型是 Delphi1.0 中字符串的类型，ShortString 类型有时又被称为 Pascal 字符串（Pascal String）或长度-字节字符串（length-byte-string）。$H 编译开关的值用来决定当变

量声明为字符串时，它是被当作 ShortString 类型还是 AnsiString 类型。在内存中，字符串就是一个字符数组，在字符串的第 0 个元素中存放了字符串的长度，紧跟在后面的字符就是字符串本身。

ShortString 缺省的最大长度为 256 个字节，这表示在 ShortString 中不能有大于 255 个字符（255+1=256）。相对于 AnsiString 来说，用 ShortString 是相当随意的，因为编译器会根据需要为它分配空间，所以不用担心中间结果是不是预先分配内存。

当用数组的下标来访问 ShortString 中的一个特定字符时，如果下标的索引值大于声明时 ShortString 的长度，则会得到假的结果或造成内存混乱。

（3）WideString 类型

WideString 类型像 AnsiString 一样是生存期自管理类型，它们都能动态分配、自动回收并且彼此能相互兼容，不过 WideString 和 AnsiString 的不同主要在三个方面：

- WideString 由 WideChar 字符组成，而不是由 AnsiChar 字符组成的，它们跟 Unicode 字符串兼容。
- WideString 用 SysAllocStrLen() A P I 函数进行分配，它们跟 OLE 的 BSTR 字符串相兼容。
- WideString 没有引用计数，所以将一个 WideString 字符串赋值给另一个 WideString 字符串时，就需要从内存中的一个位置复制到另一个位置。这使得 WideString 在速度和内存的利用上不如 AnsiString 有效。

（4）以 NULL 结束的字符串

正如前面所提到的， Delphi 有三种不同的以 null 结束的字符串类型： PChar、PAnsiChar 和 PWideChar。它们都是由 Delphi 的三种不同字符组成的。这三种类型在总体上跟 PChar 是一致的。PChar 之所以保留是为了跟 Delphi 1.0 和 Win32 API 兼容，而它们需要使用以 null 结束的字符串，PChar 被定义成一个指向以 null（零）结束的字符串指针(如果对指针的概念不熟悉，请继续读下去，在本章的后面部分会详细地介绍。与 AnsiString 和 WideString 类型不同，PChar 的内存不是由 Object Pascal 自动产生和管理的，要用 Object Pascal 的内存管理函数来为 PChar 所指向的内存进行分配。PChar 字符串的理论最大长度是 4GB。

（5）字符串运算符

可以使用+运算符或 Concat()函数来连接两个字符串，推荐使用+运算符，因为 Concat()函数主要用来向后兼容，下面程序演示了+运算符和 Concat()函数的用法：

```
{使用+运算符}
var
   s1,s2:string;

begin
   s1 := 'Hello';
   s2 := 'World';
   s1 :=s1+s2;     //'Hello World'
end.

   {用Concat() }
```

```
var
   s1,s2:string;

begin
   s1 := 'Hello';
   s2 := 'World';
   s1 :=Concat(s1,s2);   //'Hello World'
end.
```

可以使用 Delphi 丰富的运算符、过程和函数来处理字符串型的变量和属性。下面介绍几个其它常用的运算符和 Delphi 的过程或函数：

- Copy 会返回一个字符串中的子字符串；
- Delete 在一个字符串中从一个指定位置起删除一定数目的字符；
- Insert 在一个字符串中插入一个字符串；
- Length 返回字符串的长度；
- Pos 返回一个子字符串在一个字符串中的位置，即索引值。

在 SysUtil 单元中有许多函数和过程用来使字符串更易使用，请查阅 Delphi 的联机帮助“字符串处理例程”。

3. 结构类型（Struct）

简单数据类型包括集合类型，数组类型，记录类型，文件类型，类类型，类引用类型，接口类型。如图 2-2 所示。

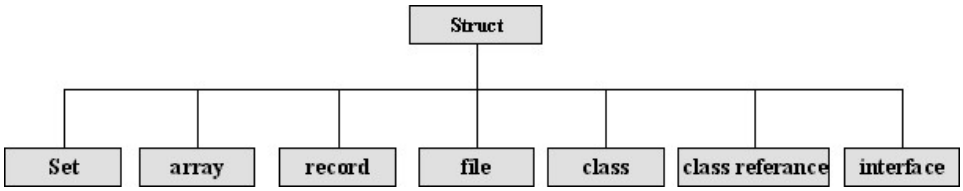

图 2-2 结构数据类型

（1）集合类型（set）

集合类型是一群相同类型元素的组合，这些类型必须是有限类型如整形、布尔型、字符型、枚举型和子界型。在检查一个值是否属于一个特定集合时，集合类型非常有用。

集合类型的定义方法：set of BaseType。例如：

```
type
    TInt = 0..255;          //BaseType为有序类型
    T1 = set of TInt;       //定义集合类型
    Tdate = set of(Wed,Mon,Thu,Sun,Sat);
    TChar = set of（'a', 'b', 'c'）;
```

Object Pascal 提供了几个用于集合的运算符，用这些运算符可以判断集合与集合之间的关系，对集合增删元素，或对集合进行求交集运算等。

（a）关系运算

用 in 运算符来判断一个给定的元素是否在一个集合中，下面的代码判断在前面所定义的集合 T1 中是否有 200：

```
if 200 in T1 then
/ /继续运行
下面的代码判断在TDate中是否没有Mon :
if not (Mon in TDate) then
/ /继续运行
```

（b）增删元素

用+、-运算符或 Include()和 Exclude()过程，可以对一个集合变量增删元素：

```
Include(T1, 256); / /在集合中增加256 ;
CharSet : = CharSet + ['d'] ; / /在集合中增加'd' ;
Exclude(CharSet, 'a'); / /在集合中删除'a' ;
CharSet:=CharSet-['a', ' b']; / /在集合中删除'a', ' b' ;
```

尽可能地用 Include()和 Exclude()来增删元素，而少用＋、－运算符。因为 Include()和 Exclude()仅需要一条机器指令，而＋和－需要 13+6n（n 是集合的按位的长度）条机器指令。

（c）交集

用*运算符来计算两个集合的交集，表达式 Set1*Set2 的结果是产生的集合的元素在 Set1 和 Set2 集合中都存在，下面的例子用来判断在一个给定的集合中是否有某几个元素：

```
if {'a', 'b', 'c'}*CharSet={'a', 'b', 'c'} then
//继续程序
```

（2）数组类型（array）

数组类型是某种数据类型的有序组合，其中每一个元素的值由其相对位置来指定，您可以在数组的某个位置上放置数据，并在需要时使用这些数据。下面的类型声明了一个 Double 型的数组变量：

```
var
  TDou : array [1..25] of Double;
```

它表示 TDou 指向一个含有 25 个 Double 型元素的数据串列，代表每一个元素的是 1 到 10 之间的数字，称为索引。数组的每一项由数组名称加上[]中的索引来表示。TDou 包含 10 个变量，TDou [1]表示第一个变量。也可以把数组定义成类型：

```
type
  Dou = array[1..25] of Double;
```

则变量说明改为：

```
var
  TDou : Dou;
```

用户可以通过给数组赋值等方法来使用数组。下面的语句将 0.0 赋给 Check 数组中的所有元素：

```
for i := 1 to 25 do
   TDou [i] := 0;
```

数组也可以是多维的，下面的类型定义了一个 20 行、20 列的数组。

```
type
   TTable = array[1..25,1..25] of Double;
var
table1:TTable;
```

想将这一表格的所有数据初始化为 0.0，您可以使用 for 循环：

```
var
Col,Row:Integer;
…
for Col :=1 to 25 do
   for Row := 1 to 25 do
Table1[Col,Row] := 0.0;
```

（3）记录类型（record）

记录是您的程序可以成组访问的一群数据的集合。记录类型中下面的例程说明了一个记录类型的用法：

```
type
   TEmployee=record
   Name : string[20];
   YearHired:1990..2000;
   Salary: Double;
   Position: string[20];
end;
```

记录包含可以保存数据的域，每一个域有一个数据类型。上文的记录 TEmployee 类型就含有四个域。您可以用以下的方式说明记录型的变量：

```
var
   NewEmployee,PromotedEmployee:TEmployee;
```

用如下的方法可以访问记录的单域：

```
NewEmployee.Salary := 1000;
```

编写如下的语句可以给整个记录赋值：

```
    with PromotedEmployee do
     begin
        Name := 'Wang Hong';
        YearHired := 1993;
```

```
    Salary := 2000.00
    Position := 'editor';
  end;
```

在程序里可以将记录当成单一实体来操作：

```
PromptEmployee := NewEmployee;
```

以上介绍了用户常用的自定义类型。在 Delphi 的编程中，对象是非常重要的用户自定义数据类型。象记录一样，对象是结构化的数据类型，它包含数据的域（Field），也包含作为方法的过程和函数。在 Delphi 中，当用户向窗体中加入一个部件，也就是向窗体对象中加入了一个域；每一个部件也是对象，每当用户建立一个事件处理过程使得部件可以响应一个事件时，用户即自动地在窗体中加入了一个方法。在本章第 2 节中，将详细讲述 Delphi 面向对象编程的方法和技巧。

文件类型（file），类类型（class），类引用类型（class referance），接口类型（interface）等，将在后面的章节中进行介绍。

4. 指针类型（pointer）

指针类型的变量指向内存空间的地址。定义形式如下：

```
type PointerName = ^typt;
```

下面的例子对指针的定义和使用进行了说明。

```
program Project1;
{$APPTYPE CONSOLE}

type
  Pint=^integer;            //定义指针类型

var
  a:integer;
  b:integer;
  pt:Pint;                  //整形指针
  p:pointer;                //无类型指针

begin
  a:=2;
  b:=3;
  pt:=@b;                   //整形指针指向整形数据
  writeln('b=',pt^);

  p:=@a;                    //无类型指针指向整形数据
  // b:=p^;                 //不可以直接赋值给变量
  b:=integer(p^);           //无类型指针指向的内容给整形变量赋值
  writeln('b=',b);

  pt:=p;                    //指针间赋值
  writeln('pt^:',pt^);
```

```
  readln;
end.
```

运行结果如下：

```
b=3
b=2
pt^:2
```

程序说明：

- 将@运算符放在变量前面，将获取变量的地址，并可以把地址赋值为同样数据类型的指针。把^运算符放在一个数据类型的前面，可以定义该类型的一个指针类型；如果放在一个指针的后面，可以获取该指针指向的地址空间的内容。
- 例子中定义了一个无类型的指针 p，它可以指向任何类型的数据，但使用过程中要注意类型转换，不可以直接将所指地址中的内容直接赋值给其它类型变量。

为了方便使用者，在 Object Pascal 语言中已经定义好了一些标准指针，程序员可以直接使用。

表 2-3 Object Pascal 语言中的一些标准指针

指针类型	所指变量的类型
Pstring,PAnsiString	AnsiString
PbyteArray	ByteArray
Pcurrecy	Currency
Pextended	Extended
PoleVariant	OleVariant
PshortString	ShortString
Pvariant	Variant
PvarRec	TvarRec(定义在 System 单元中)
PtextBuf	TextBuf(定义在 SysUtils 单元中)
PwideString	WideString
PwordArray	TWordArray(定义在 SysUtils 单元中)

5. 过程和函数类型（procedural）

过程和函数类型将在后面的 2.1.8 节中介绍。

6. 变体类型（variant）

从 Delphi 2.0 开始引进了一个功能强大的数据类型，称为变体类型(Variant)，主要是为了支持 OLE 自动化操作。Delphi 3.0 引进了一个新的被称为 Ole Variant 类型，它跟 Variant 基本一致，但是它只能表达与 OLE 自动化操作相兼容的数据类型。

有时候变量的类型在编译期间是不确定的，而 Variant 能够在运行期间动态地改变类型，这就是引入 Variant 类型目的。

Variant 能支持所有简单的数据类型，例如整型、浮点型、字符串、布尔型、日期和时间、货币以及 O L E 自动化对象等。注意 Variant 不能表达 Object Pascal 对象。Variant 可以表达不均匀的数组(数组的长度是可变的，它的数据元素能表达前面介绍过的任何一种类型，也可包括另一个 Variant 数组)。

7. 强制类型转换和类型约定

强制类型转换是一种技术，通过它能使编译器把一种类型的变量当作另一种类型。由于 Pascal 有定义新类型的功能，因此编译器在调用一个函数时对形参和实参类型匹配的检查是严格的。因此为了能通过编译检查，经常需要把一个变量的类型转换为另一个变量的类型。例如，假定要把一个字符类型的值赋给一个 byte 类型的变量，相应代码如下所示：

```
var
      c:char;
      b:byte;
begin
      c:= 's';
      b:=c; // 编译器要提示错误
end.
```

在下面的代码中，强制类型转换把 c 转换成 byte 类型，事实上强制类型转换是告诉编译器“我知道我正在干什么”，并要把一种类型转换为另一种类型：

```
var
   c:char;
   b:byte;
begin
   c:= 's';
   b:=byte(c);     // 编译器不报错
end.
```

注意：

只有当两个变量的数据长度一样时，才能对变量进行强制类型转换。

例如，不能把一个 Double 强制类型转换成 Integer。为了把一个浮点数类型转换为一个整型，要用 Trunc()或 Round()函数。为了把整型转换成一个浮点数类型的值，用下列的赋值语句：

```
floatVar :=intVar ;
```

2.1.6 运算符(Operators)

运算符是在代码中对各种数据类型进行运算的符号。例如，有能进行加、减、乘、除的运算符，有能访问一个数组的某个单元地址的运算符。

按照操作数数目来分，运算符可以分为：单目运算符（Unary Operator）和双目运算符（Binary Operator）。

有些运算符要根据给定的操作数的数据类型来作相应处理。例如，运算符 not 对于整形的操作数来说，做的是按位取反；对于逻辑类型的操作数来说，做的是逻辑取反。

下面将主要结合与 C/C++，Visual Basic 的运算符规则，介绍 Object Pascal 运算符。

1. 赋值运算符

对于 Pascal 的新手，Delphi 的赋值运算符可能是最不习惯的事情之一。为了给一个变量赋值，需要用:=运算符，而不像在 C 中用=运算符。Pascal 程序员称它为获得运算符或赋值运算符，下列表达式：

```
Number：=5；
```

可以读成 Number 获得值 5，或 Number 被赋值为 5。

2. 比较运算符

Object Pascal 用=运算符对两个表达式或两个值进行逻辑比较运算， Object Pascal 中的=运算符跟在C语言中的= =运算符相同，所以在C语言中的表达式 if(x==y) 在Object Pascal 中写成 if x=y。

注意在 Object Pascal 中:=运算符式用来为一个变量赋值，而=运算符是比较两个操作数的值。Delphi 的不等于运算符是<>，相当于 C 语言中的！=运算符。

3. 逻辑表达式

Pascal 用单词 and 和 or 作为逻辑与和逻辑或运算符，而 C 语言使用&&和||作为逻辑与和或的运算符。与和或最常用的是作为 if 语句或循环语句的一部分，就像下面两个例子演示的：

```
if A and B then
//执行下面的程序

while A or B do
//执行下面的程序
```

Pascal 的逻辑非的运算符是 not，它是用来对一个布尔表达式取反，相当于 C 语言中的!运算符，它同样也经常作为 if 语句的一部分，例如：

```
if not A then
//执行下面的程序
```

表 2-4 是一个简明参考表，列出了 Object Pascal、C/C++和 Visual Basic 的运算符。

表 2-4 赋值、比较和逻辑运算符

运算符	Pascal	C/C++	Visual Basic
赋值	:=	=	=
比较	=	==	=或 is
不等于	<>	!=	<>
大于	>	>	>

（续表）

运算符	Pascal	C/C++	Visual Basic
小于	<	<	<
小于	<	<	<
小于等于	<=	<=	<=
逻辑与	And	&&	And
逻辑或	Or	||	Or
逻辑非	Not	!	No

4. 算术运算符

表 2-5 列出了所有 Object Pascal 算术运算符以及跟 C、Visual Basic 的对照。

表 2-5 算术运算符

运算符	Pascal	C/C++	Visual Basic
加	+	+	+
减	-	-	-
乘	*	*	*
浮点数除	/	/	/
整数除	div	/	/
取模	Mod	%	Mod
指数	无	无	^

你可能注意到了 Pascal 和 Visual Basic 对浮点数和整型数除法运算用了不同的除法运算符，这点不像在 C/C++语言中。在两个整数相除时，d i v 运算符自动截取余数取整。注意对不同类型的表达式相除要选用不同的除法运算符，如果用 div 运算符对两个浮点数或用/运算符对两个整型数进行除法运算，Object Pascal 编译器将提示出错了，请看下面的代码：

```
var
    I:integer;
    F:real;
begin
    I:=6/5;              //将会引起编译错误
    F:=2.5 div 1.6;      //将会引起编译错误
end.
```

许多其它语言不区分浮点数和整数相除，而是进行浮点数相除，然后根据需要将结果转化为整型数，从性能上看，这可能开销要大一些，而 Pascal 的 div 运算符更快、更有效。

5. 位运算符

表 2-6 按位运算符

运算符	Pascal	C/C++	Visual Basic
与	And	&	and
或	Not	~	not
取反	Or	|	or
异或	Xor	^	xor
左移	Shl	<<	无
右移	Shr	>>	无

按位运算符能修改一个变量的单独各位。最常用的按位运算符能把一个数左移或右移，或对两个数按位执行与、取反、或和异或运算。移位运算符 s h l 和 s h r 分别相对于 C 语言中的< <和> >运算符，表 2-6 列出这些按位运算符。

6. 加减运算符

加减运算过程用来对一个给定的整型数执行加 1 或减 1 运算，这是经过优化的代码，Pascal 不提供像 C++中的++和--等类似的运算符，但 Pascal 提供了 Inc()和 Dec()来执行相同的功能。用一个或两个参数来调用 Inc()或 Dec()，例如，下面的代码分别对 variable 执行加 1 和减 1 操作：

```
Inc(variable);
Dec(variable);
```

同上面相比较，用 Inc()和 Dec()分别对变量执行加 3 和减 3 操作：

```
Inc(variable,3);
Dec(variable,3);
```

表 2-7 列出了在不同语言中执行加减运算的运算符。

表 2-7 加减运算符

运算符	Pascal	C/C++	Visual Basic
加	Inc()	++	无
减	Dec()	--	无

注意如果允许编译优化，Inc()和 Dec()过程就产生如下的机器码语法：variable:=variable+1; 用这样的方法对变量进行加减运算，可能感到更方便。

7. 指针运算符

表 2-8 介绍了指针运算符。

表 2-8 Object Pascal 语言中的指针运算符

运算符	作用	操作数类型	结果
+	将指针指向的地址增加偏移量	指针，integer	指针
-	将指针指向的地址减去偏移量	指针，integer	指针，integer
^	取指针所指向地址中的内容	指针	指针指向数据类型
=	判断两个指针是否指向同一地址	指针	Boolean
<>	判断两个指针是否指向不同地址	指针	Boolean

8. 集合运算符

集合运算符主要对两个集合进行操作，判断两个集合之间的关系，如表 2-9 所示：

表 2-9 Object Pascal 语言中的集合运算符

运算符	作用	操作数类型	结果
+	集合并	Set	Set
-	集合差	Set	Set
*	集合交	Set	Set
<=	左边集合是否包含于右边集合	Set	Boolean
>=	左边集合是否包含右边集合	Set	Boolean
=	左边集合是否等于右边集合	Set	Boolean
<>	左边集合是否不等右边集合	Set	Boolean
In	左边集合是否与右边集合有从属关系	有序,Set	Boolean

9. 类运算符

类运算符 as 和 in 对类或类的实例进行操作.此外,关系运算符=和<>也可以对类进行操作。

10. @运算符

@运算符返回一个变量，过程或函数的地址。

2.1.7 语句

在 Object Pascal 语言中，主要有下面一些语句：

1. 常量声明语句

常量在说明时就被赋予了一个值，在程序执行过程中是不可改变的。下面的例子说明了三个常量：

```
const
    Pi = 3.14159;
    Answer = 342;
    ProductName = "Delphi";
```

象变量一样，常量也有类型。不同的是，常量假设其类型就是常量说明中其所代表的值

的类型。上文的三个常量的类型分别是 real 型、整形、字符串型。常量用“= " 表示两边的值是相等的。

2. 赋值语句

赋值语句的形式如下：

```
variable：=expression
```

3. goto 语句

goto 语句可以从程序中的一个地方直接跳转到另一个地方。但是，从结构化程序设计的角度来考虑，尽量不要使用 goto 语句。

goto 语句的形式如下：

```
  goto label
```

在使用 goto 语句之前，首先要声明标号。标号声明语句如下：

```
label label1，label2，……，labeln；
```

4. 复合语句

首尾使用 begin 和 end 包括起来的一组语句称为复合语句。复合语句可以嵌套使用，也允许空的复合语句。例如：

```
begin
  c：=a；
  a：=b；
  b：=c；
  begin
  end；
end；
```

5. if 语句

if 语句的形式如下：

```
if A then B
if A then B else C
```

if 语句会计算一个表达式，并根据计算结果决定程序流程。if 保留字后跟随一个生成 Boolean 值 True 或 False 的表达式。一般用“=”作为关系运算符，比较产生一个布尔型值。当表达式为 True 时，执行 then 后的语句。否则执行 else 后的代码，if 语句也可以不含 else 部分，表达式为 False 时自动跳到下一行程序。

if 语句可以嵌套，当使用复合语句表达时，复合语句前后需加上 begin…end。else 保留字前不能加“；”，而且，编译器会将 else 语句视为属于最靠近的 if 语句。必要时，须使用 begin…end 保留字来强迫 else 部分属于某一级的 if 语句。

6. case 语句

case 语句用来在多个可能的情况中选择一个条件，而不再需要用一大堆 if..then if..then if 结构，下面的代码是 Pascal 的 case 语句形式：

```
case SelectExpression of
  caseList1:statement1;
  ……
  caseListN:statementN;
else
  statement;
end
```

注意 case 语句的选择因子必须是有序类型，而不能用非有序的类型如字符串作为选择因子。

7. repeat 语句

repeat 语句会重复执行一行或一段语句直到某一状态为真。语句以 repeat 开始，以 until 结束，其后跟随被判断的布尔表达式。参阅以下的例程：

```
i := 0;
repeat
  i := i+1;
  Writen(i);
until i=10;
```

当此语句被执行时，窗体的下方会出现 1 到 10 的数字。布尔表达式 i=10 (注意，与其它语言不同的是，“=”是关系运算符，而不能进行赋值操作)直到 repeat..until 程序段的结尾才会被计算，这意味着 repeat 语句至少会被执行一次。

8. while 语句

while 语句和 repeat 语句的不同之处是，它的布尔表达式在循环的开头进行判断。while 保留字后面必须跟一个布尔表达式。如果该表达式的结果为真，循环被执行，否则会退出循环，执行 while 语句后面的程序。

下面的程序可达到和上面的 repeat 例程同样的效果：

```
i := 0;
while i<10 do
  begin
    i := i+1;
    writeln(i);
  end;
```

9. for 语句

for 语句的程序代码会执行一定的次数。它需要一个循环变量来控制循环次数。用户需要说明一个变量，它的类型可以是整形、布尔型、字符型、枚举型或子界型。

下面的程序段显示 1 到 5 的数字，i 为控制变量：

```
var
i: integer;
……
for i := 1 to 5 do
   writeln(i);
```

以上介绍了三种循环语句。如果用户知道循环要执行多少次的话，可以使用 for 语句。for 循环执行速度快，效率比较高。如果用户不知道循环要执行多少次，但至少会执行一次的话，选用 repeat..until 语句比较合适；当用户认为程序可能一次都不执行的话，最好选用 while..do 语句。

10. if 语句

if 语句会计算一个表达式，并根据计算结果决定程序流程。在上文的例程中，根据 ColorDialog.Execute 的返回值，决定窗体的背景颜色。if 保留字后跟随一个生成 Boolean 值 True 或 False 的表达式。一般用“=”作为关系运算符，比较产生一个布尔型值。当表达式为 True 时，执行 then 后的语句。否则执行 else 后的代码，if 语句也可以不含 else 部分，表达式为 False 时自动跳到下一行程序。

if 语句可以嵌套，当使用复合语句时，前后需加上 begin…end。else 保留字前不能加“;”，而且，编译器会将 else 语句视为属于最靠近的 if 语句。必要时，须使用 begin…end 保留字来强迫 else 部分属于某一级的 if 语句。

11. Break 过程

在 while、for 或 repeat 循环中调用 Break()，使得程序的执行流程立即跳到循环的结尾，在循环中当某种条件满足时需要立即跳出循环，这时调用 Break()。下面的代码演示了在 30 次循环后跳出循环。

```
var
      i:integer;
begin
      for i:=1 to 100 do
      begin
            if i=30 then break;
      end;
end;
```

12. Continue( )过程

如果想跳过循环中部分代码重新开始下一次循环，就调用 Continue()过程。注意下面的例子在执行第二次循环时 continue()后的代码不执行：

```
var
   i:integer;
begin
```

```
  for i:=1 to 3 do
begin
  writeln(i, 'before continue');
  if i=2 then continue;
  writeln(i, 'after continue');
end;
  end;
```

13. With 语句

使用记录类型的变量时，可以通过 with 语句指定一些语句都是针对某一个变量来说的，这样可以简化代码的输入量。With 语句的形式如下：

```
with obj do statement
```

具体举例如下：

```
type
        TEmployee=record
          Name : string[20];
          YearHired:1990..2000;
          Salary: Double;
          Position: string[20];
      end;
       var
          employee：Temployee
          ……
       with employee do
       begin
          Name ：='wang';
          YearHired :=2001;
          Salary :=10000;
          Position := ' Technic Department';
        end;
```

在后面介绍了类和对象后，对于对象也可以使用 with 语句。

2.1.8 过程与函数

过程与函数是实现一定功能的语句块，是程序中的特定功能单元。可以在程序的其它地方被调用，也可以进行递归调用。过程与函数的区别在于：过程没有返回值，而函数有返回值。

1. 过程与函数的定义

过程与函数的定义包括过程原型或函数原型、过程体或函数体的定义。过程定义的形式如下：

```
procedure ProcedureName(ParameterList); directives;
var
LocalDeclarations;
begin
statements
end;
```

ProcedureName 是过程名，是有效的标识符。ParameterList 为过程的参数列表，需要指明参数的个数和数据类型。directives 是一些关于函数的指令字，如果设置多个，应该用分号隔开。LocalDeclarations 中定义了该函数中需要使用的一些临时变量，通常也称作本地变量。在 begin 与 end 之间是在过程调用时实现特定功能的一系列语句。ParameterList、directives、LocalDeclarations 和 statements 都是可选部分，可以不要。

函数的定义与过程非常类似，只是使用的保留字不同，而且多了一个返回值类型。具体形式如下：

```
function FunctionName(ParameterList): ReturnType; directives;
var
LocalDeclarations;
begin
statements
end;
```

可以将函数要返回的数值赋值给变量 Result。如果函数体中存在着一些由于判断而产生的分支语句时，要在每一个分支中设置返回值。通常要根据函数的返回值来确定下一步的操作。注意这里与 C 和 C++不一样，把一个值赋给 Result，函数并不会结束。

2．参数

函数定义时参数列表中的参数称为形参，将函数调用时参数列表中的参数称为实参。在定义的函数原型中，多个参数之间用分号隔开，同一类型的参数可以放在一起，以逗号隔开。在函数调用的时候，在函数原型中，多个参数之间用逗号隔开。

一般来说，形参列表和实参列表完全匹配是指参数的个数一样，而且顺序排列的数据类型也完全一致。对于普通的函数，如果编译器发现实参的数据类型与形参的数据类型不匹配，会将实参的数据类型进行一次或多次的“提升”，比如：将 integer 类型转换为 double 类型。可以为过程和函数的参数指定默认数值。

指定默认数值的参数要放在参数列表的后部，将没有指定默认数值的参数放在参数列表的前面部分。在函数调用的时候，可以为设置了默认值的参数指定一个新值，在函数体中，各语句使用的是指定的新值；如果没有指定新值，则使用默认值。同样，如果存在多个设置了默认值的参数，只有前面的参数指定了新值，后面的参数才可以指定新值。

下面的例子定义了一个函数 OutputNum，可以将一个浮点数按指定的精度输出。通过这个例子，读者可以体会函数中参数的使用。

```
program Project1;
{$APPTYPE CONSOLE}
```

```
uses Sysutils; // 为了使用函数Format
function OutputNum(number:double;n:integer = 5):Boolean;
var
Str : string; // 浮点数显示输出的内容
begin
if n <= -1 then // 小数点后的位数要大于或等于零
begin
Result: =False;
Exit; // 退出显示函数
end
else
begin
// 设置显示的格式
Str := Format('%*.*f', [10, n, number]);
Result : = True ;
Writeln(Str); // 显示数据
end;
end;
begin
OutputNum(12.345); // n默认为5
OutputNum(123,3); // 参数对数据类型进行升级
// 下面一句代码不正确，故屏蔽掉
// OutputNum(123.456789,9.13); // 参数对数据类型不能降级
// 可以根据函数的返回值确定下一步的操作
if OutputNum(123.456789,-3) = false then
Writeln( '输出失败。' ) ;
Readln;
end.
```

运行结果如下：

```
12.34500
123.000
```

输出失败。

这里有几点需要说明：

- 为了使用函数 Format，需要在 uses 语句中将 Sysutils 单元包含进去。
- 由于小数点后的位数不可以设置为负数，所以当出现负数的时候， OutputNum 函数返回 False，并调用 exit 函数立刻退出 OutputNum 函数。
- 在语句 OutputNum(123,3);中，首先将整型常数 123 转换为浮点型常数，然后进行参数传递。

最常用的参数有数值参数、变量参数和常量参数三种。

数值参数在运行过程中只改变其形参的值，不改变其实参的值，即参数的值不能传递到

过程的外面。试看下面的例程：

```
procedure calculate(CalNo:Integer);
begin
   CalNo := CalNo*10;
end;
```

用以下例程调用 calculate 函数：

```
calculate(Number);
```

Number 进入 Calculate 函数后，会把 Number 实参拷贝给形参 CalNo，在过程中 CalNo 增大十倍，但并未传递出来，因此 Number 值并未改变。形参和实参占用不同的内存地址，在过程或函数被调用时，将实参的值复制到形参占用的内存中。因此出了过程或函数后，形参和实参的数值是不同的，但实参的值并不发生变化。

如果用户想改变传入的参数值，就需要使用变量参数，即在被调用程序的参数表中的形参前加上保留字 var。例如：

```
procedure Calculate(var CalNo : Integer);
```

则 CalNo 并不在内存中占据一个位置，而是指向实参 Number。当一个变参被传递时，任何对形参所作的改变会反映到实参中。这是因为两个参数指向同一个地址。将上一个例程中过程头的形参 CalNo 前面加上 var，再以同样的程序调用它，则在第二个编辑框中会显示计算的结果，把第一个编辑框中的数值放大十倍。这时形参 CalNo 和实参 Number 的值都是 Nnmber 初始值的 10 倍。

如果当过程或函数执行是要求不改变形参的值，最保险的办法是使用常量参数。在参数表的参数名称前加上保留字 const 可以使一个形参成为常量参数。使用常量参数代替数值参数可以保护用户的参数，使用户在不想改变参数值时不会意外地将新的值赋给这个参数。下面的例子将帮助读者加深理解：

```
program Project1;
{$APPTYPE CONSOLE}

type
PInteger = ^Integer; // 定义指针类型
procedure P1(var N:Integer); // 引用参数传递
begin
N:=N+1 ;
end;
procedure P2(N:Integer); // 普通参数传递
begin
N:=N+2;
end;
procedure P3(PT:PInteger); // 传递指针参数
begin
PT^:=PT^+3;
```

```
end;

var
i:Integer;
begin
i:=1;
P1(i); // 将i的值增加1
Writeln('i',i);
P2(i); // 希望将i加2，但没有实现
Writeln('i',i);
P3(@i); // 将I加3
Writeln('i':,i);
Readln;
end.
```

运行结果如下：

```
i : 2
i : 2
i : 5
```

这里有几点需要说明：

- 一开始变量 i 的数值为 1，经过 P1 过程的处理，将 i 加 1，所以显示的第一个 i 的数值为 2，这时使用的是引用参数传递。
- 在过程 P2 中，将形参的数值增加了 2，实际上 i 并没有增加，所以显示的第二个 i 的数值仍然为 2。在这种情况下，正常的做法可以使用函数的返回值，例如：

```
Result:=N+2;
```

在调用函数的时候使用：

```
i:=P2(i);
```

- 在过程 P3 中，传递的是变量 I 的指针，所以操作是针对 i 进行的，第三次显示的 i 的数值是 5。

3. 过程与函数的调用约定

在调用过程或函数的时候，如果参数列表中具有多个参数，那么参数传递给过程或函数的顺序会对结果产生一定的影响。对于不同的语言，参数传递的顺序是不同的： Pascal 语言是按照从左向右的顺序进行传递的，而 C 语言是按照从右向左的顺序来传递的。为了确定传递的顺序，可以在过程或函数定义的时候，在 directives 部分利用指令字指定传递的顺序。

表 2-10 中的数据来自 Delphi 的联机帮助，其中列举了 directives 部分可使用的关于函数调用约定的指令字。

表 2-10 定义过程与函数时对调用约定起作用的指令字

Directive 指令	Register	Pascal	Stdcall	safecall	Cdecl
参数传递顺序	从左向右	从左向右	从右向左	从右向左	从右向左

下面的例子可以通过该例程看看参数传递的顺序。

```
program Project1;
{$APPTYPE CONSOLE}
function P1:Integer; // 该函数将作为GetMax函数的第一个参数
begin
Writeln('P1');
Result=0；
end;
function P2:Integer; // 该函数将作为GetMax函数的第二个参数
begin
Writeln('P2 ') ;
Result: =1;
end;
// 参数的传递方式采用pascal方式
function Sum(N1:Integer; N2:Integer):Integer;pascal;
begin
Result:=N1+N2;
end;
begin
Sum(P1,P2);
end .
```

运行结果如下：

```
P1
P2
```

如果将 GetMax 函数定义处的 directives 部分由 pascal 改为 stdcall，则运行结果变为：

```
获取数据2
获取数据1
```

读者可以修改 GetMax 函数定义处的 directives 部分为表 2-10 中的其它数值，测试结果是否一致。

4. 过程和函数的重载

可以在同一个作用范围内给不同的过程或函数取同一个名称，这种现象就叫做重载。重载可以方便编程。重载函数必须用指令字 overload 来进行说明。在重载的情况下，决定使用哪个过程或函数的依据是形参和实参的一致性，即参数个数、参数类型以及它们之间的顺序，不存在一个函数调用满足两个重载函数的情况。另外，函数的返回值类型不同不可以作为重

载函数的依据。下面的两个函数就是重载函数：

```
function Average(a:Integer; b:Integer):Double;overload; // 求整形数据的平均值
function Average(a:Double; b:Double):Double;overload; // 求实数数据的平均值
```

下面两条语句就调用了不同的函数：

```
Average(3.7,4.6); // 调用的是第二个重载函数
Average(3,4); // 调用的是第一个重载函数
```

如果又定义了一个重载函数如下：

```
function Average(a,b:Double;c:Double=0.0):Double;overload; // 求三个实数平均值
```

尽管参数的个数与上面的两个不同，但第三个参数设置了一个默认值，当参数调用为语句 Average(1.1,2.2);时，编译系统就不知道应该使用哪个重载函数了，因为第二个重载函数和第三个重载函数都可以满足要求，所以会出现一个编译错误。

5．函数的递归调用

在 Object Pascal 中，过程或函数必须先说明再调用。以上规则在递归调用时是例外情况。所谓递归调用，是指函数 A 调用函数 B，而函数 B 又调用函数 A 的情况，或是指一个函数调用自身的特殊情况。在递归调用中，函数要进行前置，即在函数或过程的标题部分最后加上保留字 forword。下文的例子是一个递归调用的典型例子：

```
program Project1;
{$APPTYPE CONSOLE}
    var
      alpha:Integer;
    procedure Test2(var A:Integer):forword;
      {Test2被说明为前置过程}
    procedure Test1(var A:Integer);
    begin
      A :=A-1;
      if A>0 then
        test2(A);     {经前置说明，调用未执行的过程Test2}
      writeln(A);
    end;
    procedure Test2(var A:Integer);{经前置说明的Test2的执行部分}
    begin
      A :=A div 2;
      if A>0 rhen
        test1(A);   {在Test2中调用已执行的过程Test1}
    end;
    begin
```

```
    Alpha := 15;   {给Alpha赋初值}
    Test1(Alpha); { 第一次调用Test1,递归开始}
end;
```

程序开始时给 Alpha 赋初值，并实现先减 1 再除 2 的循环递归调用，直到 Alpha 小于 0 为止。

2.1.9 作用范围

1. 标识符的作用范围

一个变量、常量、方法、类型或其它标识符的范围定义了这个标识符的活动区域。对于说明这个标识符的最小程序模块而言，此标识符是局部的。当用户的应用程序在说明一个标识符的程序模块外执行时，该标识符就不在此范围内。这意味着此时执行的程序无法访问这个标识符，只有当程序再度进入说明这个标识符的程序模块时，才可以访问它。

图 2-3 表示一个含有两个库单元的工程，每个库单元中又各有三个过程或事件处理过程。

图中每一个矩形代表一个程序模块。如果用户在程序 D 中说明一个变量，则只有过程 D 可以访问它，它是 D 的局部变量。如果用户在程序 B 中说明一个不属于任何过程的变量，那么过程 D、E、F 都能使用它，因为以上的过程都属于库单元 B，因此这个变量对以上过程是全局的，但对于程序库单元 B 它是局部的。一般来讲，应尽量把标识符说明成局部的，这样会增加程序对数据的保护，使它不会被不小心修改。只有当几个程序模块分享数据时，才需要考虑使用全局的说明。

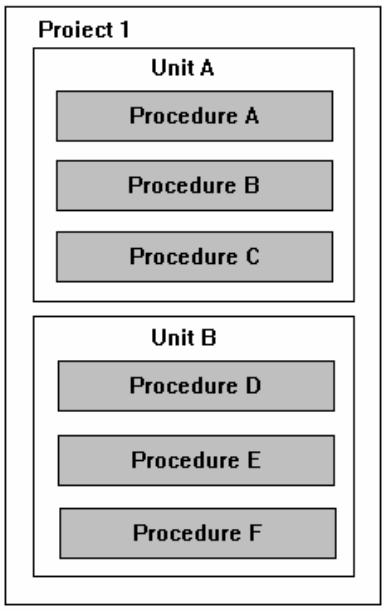

图 2-3 一个简单过程中的程序模块

2. 访问其它程序模块中的说明

用户可以在当前的程序模块中访问其它程序模块中的说明。例如用户在库单元中编写一个事件处理过程来计算利率，则其它的库单元可以访问这个事件处理过程。要访问不在当前库单元中的说明，应在这个说明之前加上其它应用程序的名称和一个点号(.)。例如，在库单元 Unit1 中有事件处理过程 CalculateInterest 过程，现在用户想在库单元 Unit2 中调用这一过程，则可以在 Unit2 的 uses 子句中加入 Unit1，并使用下面的说明：

```
Unit1.CalculateInterest(PrincipalInterestRate : Double);
```

应用程序的代码不能在一个模块外访问它说明的变量。事实上，当程序执行跳出一个模块后，这些变量就不存在于内存中了。这一点对于任何标识符都是一样的，不管事件处理过程、过程、函数还是方法，都具有这一性质。这样的标识符称为局部变量。

2.1.10 规范化命名

在系统开发的过程中，常常要为变量、类、对象、函数和文件等命名。一般在开发的需求或设计阶段就必须制定出一套完整、实用的命名规则。这样，在很大程度上可以提高系统开发的效率，便于不同模块之间的接口，方便系统的维护。

在制定命名规则的时候，一个基本的原则就是便于使用、便于维护、风格统一。应该注意下面几点：

- 命名时要采用英文单词，而不要使用中文拼音，尤其不要使用中文拼音第一个字母的组合。在使用英文单词命名时，尽量采用统一、简单、贴切的词语，尽可能使用完整的单词或音节。
- 有些名称可以采用几个英文单词的组合。在组合过程中，尽量不要使用下划线来分隔单词，最好采用大小写混写的方式来实现。
- 对于保留字和指令字可以统一全部小写，而对于一些常量名可以全部大写。
- 有些名称可以是“动词+对象”组合而成，也可以是“对象+动词”组合而成。一般来说，“动词+对象”比较符合平常的语法习惯。但不管怎样，整体上都应该统一。
- 在有些情况下，要考虑到与 Delphi 集成开发环境的统一。例如，在 Delphi 集成开发环境中，普通类的名称一般以 T 开头，异常类的名称一般以 E 开头。
- 在对菜单命令的标识号命名的时候，应将所属菜单项的名称包含进去。比如对于“文件”菜单项中的菜单命令，可以将标识号命名为 FileOpen、FileClose 等。
- 对于一些表示集合意义的名称，可以使用名词的复数形式。比如窗口的集合，可以使用 Windows，而不要使用 WindowCollection。

2.2 Object Pascal 语言的面向对象技术

软件系统开发过程中，结构分析和结构设计技术具有很多优点，但是也存在着许多难以克服的缺点。因为结构分析和结构设计技术是围绕着实现处理功能来构造系统的，而在系统维护、软件升级的过程中，用户的需求变化往往是针对系统功能，所以，采用这种技术设计的系统是不稳定的，其可修改性和重用性都非常差。

在这种情况下，面向对象的程序设计技术产生了，它尽可能地模拟人类习惯的思维方式，使开发软件的方法和过程尽可能地接近人类认识世界、解决问题的方法与过程。采用面向对象的程序分析和设计技术开发的软件系统，稳定性、可重用性和可维护性都很好，后面将进行具体说明。

Delphi 是基于面向对象编程的先进开发环境。面向对象的程序设计(OOP)是结构化语言的自然延伸。OOP 的先进编程方法，会产生一个清晰而又容易扩展及维护的程序。一旦用户为用户的程序建立了一个对象，用户和其它的程序员可以在其它的程序中使用这个对象，完全不必重新编制繁复的代码。对象的重复使用可以大大地节省开发时间，切实地提高用户和其它人的工作效率。

OOP 是使用独立对象(包含数据和代码)作为应用程序模块的范例。虽然 OOP 不能使得代码容易编写，但它能使代码易于维护。将数据和代码结合在一起，能使定位和修复错误的工作得到简化，并最大限度地减少了对其它对象的影响，提高了程序的性能。对于面向对象，Coad 和 Yourdon 给出了下面直观的定义：

面向对象 = 对象 + 类 + 继承 + 通信

2.2.1 对象和类的概念

客观世界中，每一个有明确意义和边界的事物都可以看作是一个对象(Object)，这些对象有自己的属性，对象与对象之间还有一定的相互关系。

例如，日常生活中使用的家用电器就可以看作是一个对象，它们都需要接通电源，都可以完成一定工作。不同种类的家用电器又可以看成不同的对象，比如：电视机，电冰箱，洗衣机，录像机等。不同的的电器可以协同工作，比如电视机和录像机一同工作。此外，我们每个人都可以看作是一个对象，每人都有自己的个性：喜欢看电视或不喜欢看电视。我们与电视机之间还具有一定的关系：当按下电源按钮后，电视机才可以接收微波信号；同时，电视节目还可以让我们高兴、失望或者紧张等。

我们可以把具有相似特征的事物归为一类，也就是把具有相同属性的对象看成一个类(class)。比如，所有的电视机可以归成一个“电视机类”，所有的人可以归成一个“人类”。在面向对象的程序分析和设计技术中，“类”就是对具有相同属性和相同操作的一组相似对象的定义。从另一个角度来看，对象就是类的一个实例。

在程序设计阶段，一个非常重要的工作就是按照面向对象的方法去分析所要解决的问题。也就是清楚所要解决的问题中有多少实体，每一个实体所具有的属性，各个实体之间的关系等。然后把具有相同属性和相同操作的实体划分为一个类，明确每个类的属性和方法。

一个对象是一个数据类型。对象就象记录一样，是一种数据结构。按最简单的理解，我们可以将对象理解成一个记录。但实际上，对象是一种定义不确切的术语，它常用来定义抽象的事务，是构成应用程序的项目，其内涵远比记录要丰富。在本书中，对象可被理解为可视化部件如按钮、标签、表等。

了解对象，最关键的是掌握对象的特性。一个对象，其最突出的特征有三个：封装性、继承性、多态性。

1. 对象的封装性

对象最基本的理解是把数据和代码组合在同一个结构中，这就是对象的封装特性。将对象的数据域封闭在对象的内部，使得外部程序必需而且只能使用正确的方法才能对要读写的数据域进行访问。封装性意味着数据和代码一起出现在同一结构中，如果需要的话，可以在数据周围砌上“围墙”，只有用对象类的方法才能在“围墙”上打开缺口。把相关的数据和代码结合在一起，并隐藏了实现细节。封装的好处是有利于程序的模块化，并把代码和其它代码分开。

2. 对象的继承性

继承性的含义直接而且显然。是指一个新的对象能够从父对象中获取属性和方法，这种概念能用来建立 VCL 这样的多层次的对象，首先建立通用对象，然后创建这些通用对象的有专用功能的子对象。继承的好处是能共享代码，在往新对象中添加任何新内容以前，父类的每一个字段和方法都已存在于子类中，父类是创建子类的基石。

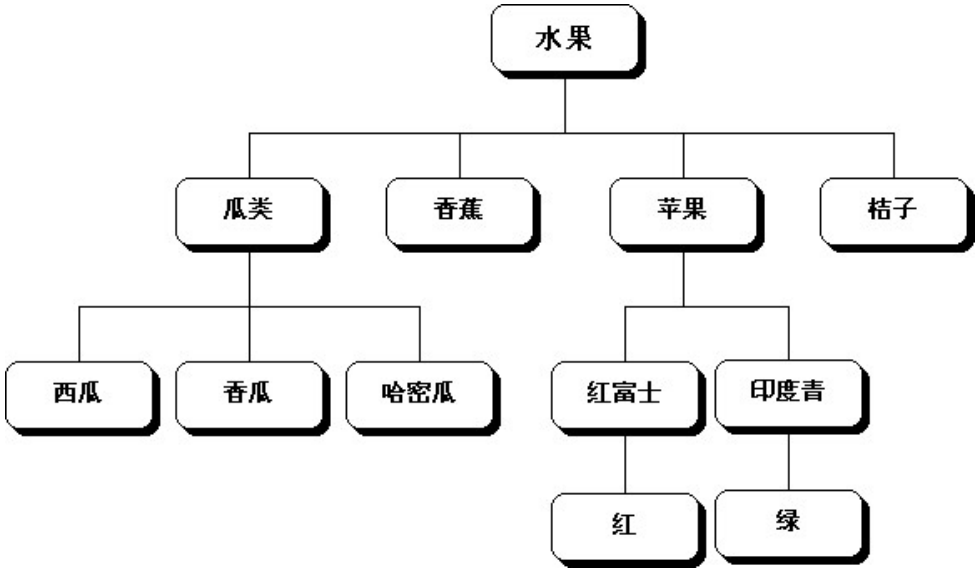

图 2-4 水果类的继承关系

图 2-4 是一个继承的例子，根结点是水果，它是所有水果的祖先；其中包括：瓜类，它是所有瓜的祖先，瓜类中包括西瓜，香瓜，哈密瓜；苹果具有不同品种对应红色和绿色的不同属性；香蕉；桔子。

如图 2-4 中，水果类是一个抽象类。在定义一个实体的时候，可能会使用一些抽象的概念来概括一类事物。比如“水果”的概念就描述了一类事物，因为可以定义一个“水果类”。但是在现实生活中，我们接触的都是某一类具体的动物，比如苹果、香蕉、瓜和桔子等。由于类具有继承性，所以可以在定义了“水果”类的基础上派生出“苹果类”、“瓜类”、“香蕉类”和“桔子类”等。同时为了避免直接创建“水果类”的实例，可以把“水果类”定义为抽象类。

3. 对象的多态性

多态性是在对象体系中把设想和实现分开的手段。如果说继承性是系统的布局手段，多态性就是其功能实现的方法。多态性意味着某种概括的动作可以由特定的方式来实现，这取决于执行该动作的对象。多态性允许以类似的方式处理类体系中类似的对象。根据特定的任务，一个应用程序被分解成许多对象，多态性把高级设计处理的设想如新对象的创建、对象在屏幕上的重显、程序运行的其它抽象描述等，留给知道该如何完美的处理它们的对象去实现。

在继续深入了解对象的概念以前，应该先理解下面三个术语：

- 域(field)，也被称为域定义或实例变量，域是包含在对象中的数据变量。在对象中的一个域就像是在 Pascal 记录中一个域，在 C++中它被称为数据成员。
- 方法(method)，属于一个对象的过程和函数名，在 C++中它被称为成员函数。
- 属性(property)，属性是外部代码访问对象中的数据和代码的访问器，属性隐藏了一个对象的具体实现的细节。

注意最好不要直接访问对象的域，因为实现对象的细节可能改变。可以用访问器属性来访问对象，它不受对象细节的影响

2.2.2 Object Pascal 中类的定义

在 Object Pascal 语言中，类和记录比较相似，也是一个构造类型，并且由属性和方法构成。其中属性又包括类的内部属性和外部属性，也就是供内部使用的一些数据变量和供外部使用的一些数据变量；方法则是该类或其实例可以操作的过程和函数。通常把类的内部属性称为字段，把字段、属性和方法统称为类的成员。

类的定义形式如下：

```
type ClassName = class (AncestorClass)
  MemberList
end;
```

上面的 ClassName 为类的名称，通常是一个以 T 开头的标识符。AncestorClass 是所继承的父类的名称。MemberList 为成员列表，可以声明一些变量和对象，也可以声明一些过程与函数。

在 Delphi 中，如果不指明父类，则默认的父类为 TObject 类，也就是直接从 TObject 类派生出一个新类。TObject 类是在 System 单元中定义的。

1. 成员属性

类的所有成员都有一个标明“能见度”的属性，它们是由保留字 private、protected、public、published 或 automated 来说明的。通过这些保留字，可以控制对类中成员的访问权限。每个保留字的具体含义如下：

（1）private

具有 private 属性的成员称为私有成员，不能被类所在单元以外的程序访问。也就是说，一个私有的属性不可以在所在模块之外的其它模块中读写，一个私有的方法也不可以在所在

模块之外的其它模块中被调用。但是，如果在同一个单元文件中定义了两个类(通常把关系非常紧密的两个类定义在同一个单元文件中)，那么，在一个类的成员中就可以对另一个类中的私有变量进行访问，或者调用另一个类中的私有方法。

（2）protected

具有 protected 属性的成员称为保护成员，可以被该类的所有派生类访问，并且成为派生类的私有成员。

（3）public

具有 public 属性的成员称为公有成员，可以被该类以外的类访问。如果两个类不在同一个单元文件中，则要在 uses 语句中包括被访问的类所在的单元名称。

（4）published

具有 published 属性的成员称为发行类型成员，它的访问权限基本与公有成员相同，只是在设计期间也可以被访问。通常发行类型的成员用在组件类的声明中，这样，就可以在对象编辑器中访问组件的发行类型的成员。

（5）automated

具有 automated 属性的成员称为自动类型成员，它的访问权限基本同公有成员。这种类型的成员一般用在从 TAutoObject 类派生的类中，目前只是为了和以前版本的 Delphi 保持兼容才保留了 automated 属性。

除了在类封装的时候可以限制成员的访问权限外，在后面介绍的单元文件中也可以限制对变量、对象、函数和过程等的访问权限。为了使软件系统具有良好的安全性、健壮性，应该注意这些限制权限的用法。

2．构造与析构

在完成了类的封装之后，就可以使用这个类了。具体的步骤如下：

- 声明类的一个变量。这时可以将类作为一种数据类型来看待。
- 调用类的一个特殊函数——构造函数来进行一些初始化工作，比如按照类的结构来分配内存资源，完成对象的创建。
- 对类的实例——对象进行操作、使用。可以修改对象的属性或调用对象的方法。
- 使用完毕，调用类的另一个特殊函数——析构函数，删除创建的对象，同时释放相应的内存资源等。此外，还可以调用 free 过程释放对象占用的资源。

构造函数和析构函数是类定义中两个非常重要的函数，它们完成的功能正好相反，它们的定义也比较特殊。在声明了类的一个变量后，并没有实际创建该类的对象，只是定义了一个指向该类对象的指针，有时也称之为类的引用。

对象的创建和初始化工作是由类的构造函数来完成的。在类的构造函数中，不仅可以根据类的结构为类的对象分配内存空间，而且还可以打开文件或数据库，读取一些初始数据，或者控制一些设备进行复位等。在定义构造函数的时候，不是使用保留字 function，而是使用保留字 constructor，通常函数名使用 Create。如果在定义类的时候没有定义构造函数，则系统会自动为该类生成一个默认的构造函数。构造函数必须使用默认的函数调用约定方式，也就是使用 register 指令字方式。

我们也可以自定义一个或多个构造函数。自定义的构造函数可以有参数列表，可以重载构造函数。一般在自定义的构造函数的函数体中，在开始部分使用 inherited 保留字来调用父类的构造函数。

如果在创建并初始化对象时，调用构造函数发生错误，则会自动调用析构函数来删除这个没有完成的对象。

析构函数的作用是将对象删除并释放相应的内存资源，此外还可以在这之前保存一些数据信息并关闭文件或数据库等，或者对一些设备进行复位并关机。在定义析构函数的时候，使用保留字 destructor 代替通常函数的 function，函数名为 Destroy。如果在定义类的时候没有定义析构函数，则系统会自动为该类生成一个默认的析构函数。析构函数也必须使用默认的函数调用约定方式，也就是使用 register 指令字方式。

我们也可以自定义析构函数。通常在自定义的析构函数的函数体中，在结尾部分使用 inherited 保留字来调用父类的析构函数。在释放对象占用的资源时也可以使用 TObject 类的成员过程 Free。使用 Free 过程可以删除一个对象，如果该对象不为 nil，则会自动调用析构函数。通常在运行时创建的对象应该调用 Free 过程来代替析构函数。因为如果对象没有被初始化，则调用析构函数时就会出错，而调用 Free 过程就没有问题。

下面的例子说明了类的定义和使用。

```
program Project1;
{$APPTYPE CONSOLE}

type
TPerson = class // 人员类
public
    Name:string; // 姓名
    function GetAge:Integer; // 获取年龄
    procedure SetAge(A:Integer); // 设置年龄
private
    Age:Integer; // 年龄
end;

TEmployee = class(TPerson) // 职员类
Public
Salary：integer; //薪金
    DeptName: string; // 部门名称
    procedure Infor; // 显示职员信息
    procedure SetSalary(A:Integer); // 设置薪水
end;

TCustomer = class(TPerson) // 顾客类
public
    AddressName: string; //地址名称
    procedure Infor; // 显示顾客信息
```

```
    constructor Create(Str: string);
    destructor Destroy;override;
end;

function TPerson.GetAge:Integer; // 获取人员的年龄
begin
    Result:=Age;
end;
procedure TPerson.SetAge(A:Integer); // 设置人员的年龄
begin
    Age:=A;
end;

procedure TEmployee.SetSalary(A:Integer); // 设置职员薪水
begin
    Salary:=A;
end;
procedure TEmployee.Infor; // 显示职员的信息
begin
    Writeln('姓名：',Name,' 年龄：',GetAge,' 部门：', DeptName，' 薪水：',Salary) ;
end;
procedure TCustomer.Infor; // 显示顾客的信息
begin
    Writeln('姓名：',Name,' 年龄：',GetAge,' 住址：', AddressName) ;
end;

constructor TCustomer.Create(Str: string); // 顾客类的构造函数
begin
inherited Create; // 调用父类的构造函数
    Writeln('顾客类的构造函数。');
    AdressName:=Str;
end;
destructor TCustomer.Destroy; // 顾客类的析构函数
begin
    Writeln ('姓名为：',Name,' 的顾客类对象被删除。') ;
    inherited Destroy; // 调用父类的析构函数
end;
var
    E1: TEmployee; // 声明一个职员类的变量
    C1: TCustomer; // 声明一个顾客类的变量
begin
    E1:=TEmployee.Create; // 调用默认的构造函数创建对象
```

```
    E1.Name:='王洪';
    E1.SetAge(21);
    E1.SetSalary(2000);
    E1.DeptName:='贷款部';
    E1.Infor; // 显示职员信息
    E1.Destroy; // 调用默认的析构函数
    C1:=TCustomer.Create('北一路'); // 调用构造函数创建对象
    C1.Name:='李伟';
    C1.Age:=46;
    // 显示顾客信息
    C1.Infor; // 显示顾客信息
    C1.Destroy; // 调用自定义的析构函数
end.
```

运行结果如下：

```
姓名：王洪 年龄：21 部门：贷款部 薪水：2000
```

调用顾客类的构造函数。
姓名：李伟 年龄：46 住址：北一路
姓名为：李伟的顾客类对象被删除。
有几点需要说明：

- 例程中的 TPerson 类默认从 TObject 类派生而来，然后它又派生出两个子类：TEmployee 类和 TCustomer 类。TEmployee 类和 TCustomer 类都继承了 TPerson 类的 Age 属性、Name 属性、GetAge 函数和 SetAge 过程等。
- TEmployee 类对 TPerson 类进行了功能扩展，又增加了 DeptName，Salary 属性和 Infor，SetSalary 过程。
- TCustomer 类对 TPerson 类功能扩展，增加了 AddressName 属性和 Infor 过程。此外，TCustomer 类中自定义了构造函数和析构函数。应该注意在自定义的构造函数和析构函数的函数体中，调用父类的构造函数和析构函数的位置。
- 调用构造函数的方法与调用析构函数的方法不同，前者使用的是“类的构造函数”，后者使用的是“对象的析构函数”。

上面例子中的人员类可以看作是一个抽象类。定义抽象方法时只要使用指令字 abstract 对其进行说明就可以了。在 Delphi 中，抽象方法一定是虚拟方法或动态方法，所以，要在指令字 abstract 前使用 virtual 或 dynamic 进行说明。此外，对于抽象方法不可以定义函数体。

我们来看一个抽象类的具体例子，相应代码如下：

```
program Project1;
{$APPTYPE CONSOLE}

type
TFruit = class // 水果类
procedure Infor;virtual;abstract;
```

```
end;
TApple = class(TFruit) // 苹果类
Name:string;
procedure Infor;override;
end;
TOrange = class(TFruit) // 桔子类
procedure Infor;override;
end;
procedure TApple.ShowInf;
begin
Writeln('苹果。');
end;
procedure TOrange.Infor ;
begin
Writeln('桔子。');
end;

var
A1: TApple; // 声明一个苹果类的变量
O1: TOrange; // 声明一个桔子类的变量
begin
A1 : = TApple.Create;
A1.Infor; // 调用的是TApple类的Infor
A1.Destroy;
O1:=TOrange.Create;
O1.Infor; // 调用的是TOrange类的Infor
O1.Destroy;
end.
```

运行结果如下：

```
苹果。
桔子。
```

2.2.3 方法

方法是属于一个给定对象的过程和函数，方法反映的是对象的行为而不是数据，刚才我们提到了对象的两个重要的方法即构造器和析构方法。为了使对象能执行各种功能，你能在对象中定制方法。

1．方法的类型

对象的方法能定义成静态(static)、虚拟(virtual)、动态(dynamic)或消息处理(message)。请看下面的例子：

```
Tfoo=class
  Procedure IamAStatic;
```

```
  Procedure IamAVitual；virtual；
  Procedure IamADynamic；dynamic；
  Procedure IamAMessage（var M：TMessage）；message wm_SomeMessage;
end;
```

（1）静态方法

IAmAStatic 是一个静态方法，静态方法是方法的缺省类型，对它就像对通常的过程和函数那样调用。编译器知道这些方法的地址，所以调用一个静态方法时它能把运行信息静态地链接进可执行文件。

静态方法执行的速度最快，但它们却不能被覆盖来支持多态性。

（2）虚拟方法

IAmAVirtual 是一个虚拟方法。虚拟方法和静态方法的调用方式相同。由于虚拟方法能被覆盖，在代码中调用一个指定的虚拟方法时编译器并不知道它的地址。因此，编译器通过建立虚拟方法表（VMT）来查找在运行时的函数地址。所有的虚拟方法在运行时通过 VMT 来调度，一个对象的 VMT 表中除了自己定义的虚拟方法外，还有它的祖先的所有的虚拟方法，因此虚拟方法比动态方法用的内存要多，但它执行得比较快。

（3）动态方法

IAmADynamic 是一个动态方法，动态方法跟虚拟方法基本相似，只是它们的调度系统不同。编译器为每一个动态方法指定一个独一无二的数字，用这个数字和动态方法的地址构造一个动态方法表(DMT)。不像 VMT 表，在 DMT 表中仅有它声明的动态方法，并且这个方法需要祖先的 DMT 表来访问它其余的动态方法。正因为这样，动态方法比虚拟方法用的内存要少，但执行起来较慢，因为有可能要到祖先对象的 DMT 中查找动态方法。

（4）消息处理方法

IAmAMessage 是一个消息处理方法，在关键字 message 后面的值指明了这个方法要响应的消息。用消息处理方法来响应 Windows 的消息，这样就不用直接来调用它。

（5）方法的覆盖

在 Object Pascal 覆盖一个方法用来实现 OOP 的多态性概念。通过覆盖使一方法在不同的派生类间表现出不同的行为。Object Pascal 中能被覆盖的方法是在声明时被标识为 virtual 或 dynamic 的方法。为了覆盖一个方法，在派生类的声明中用 override 代替 virtual 或 dynamic。例如，能用下面的代码覆盖 IAmAVirtual 和 IAmADynamic 方法：

```
TfooChild=class(TFoo)
  Procedure IamAVitual；override；
  Procedure IamADynamic；override；
  Procedure IamAMessage（var M：TMessage）；message wm_SomeMessage;
end;
```

用了 override 关键字后，编译器就会用新的方法替换 VMT 中原先的方法。如果用 virtaul 或 dynamic 替换 override 重新声明 IAmAVirtual 和 IAmADynamic，将是建立新的方法而不是对祖先的方法进行覆盖。同样，在派生类中如果企图对一个静态方法进行覆盖，在新对象中的方法完全替换在祖先类中的同名方法。

（6）方法的重载

就像普通的过程和函数，方法也支持重载，使得一个类中有许多同名的方法带着不同的参数表，能重载的方法必须用 overload 指示符标识出来，可以不对第一个方法用 overload。下面的代码演示了一个类中有三个重载的方法：

```
Type
  TSomeClass=class
     Procedure Amethod(I:integer);overload;
     Procedure Amethod(S:string);overload;
     Procedure Amethod(D:double);overload;
end;
```

（7）重新引入方法名称

有时候，需要在派生类中增加一个方法，而这个方法的名称与祖先类中的某个方法名称相同。在这种情况下，没必要覆盖这个方法，只要在派生类中重新声明这个方法。但在编译时，编译器就会发出一个警告，告诉你派生类的方法将隐藏祖先类的同名方法。要解决这个问题，可以在派生类中使用 reintroduce 指示符，下面的代码演示了 reintroduce 指示符的正确用法：

```
Type
     TSomeBase=class
     Procedure A;
end;

     TSomeClass=class
     Procedure A;reintroduce;
end;
```

（8）Self

在所有对象的方法中都有一个隐含变量称为 Self，Self 是用来调用方法的指向类实例的指针。Self 由编译器作为一个隐含参数传递给方法。

2. 属性

可以把属性看成是能对类中的数据进行修改和执行代码的特殊的辅助域。对于组件来说，属性就是列在 Object Inspector 窗口的内容。下面的例子定义了一个有属性的简单对象：

```
TMyObject=class
private
  SomeValue:integer;
  Procedure SetValue(a:integer);
Public
  Property value:integer read SomeValue write SomeValue;
end;
procedure TMyObject. SetValue(a:integer);
begin
```

```
  if SomeValue<>a then
SomeValue:=a;
end;
```

TMyObject 是包含下列内容的对象：一个域(被称为 SomeValue 的整型数)、一个方法(被称为 SetValue 的过程)和一个被称为 value 的属性。SetValue 过程的功能是对 SomeValue 域赋值，Value 属性实际上不包含任何数据。Value 是 SomeValue 域的辅助域，当想得到 Value 中的值时，它就从 SomeValue 读值，当试图对 Value 属性设置值时，Value 就调用 SetValue 对 SomeValue 设置值。这样做的好处有两个方面：首先，通过一个简单变量就使得外部代码可以访问对象的数据，而不需要知道对象的实现细节。其次，在派生类中可以覆盖诸如 SetValue 的方法以实现多态性。

在 Object Pascal 中的类实例实际上是指向堆中的类实例数据的 32 位指针。当访问对象的域、方法和属性时，编译器会自动产生一些代码来处理这个指针。因此对于新手来说，对象就好像是一个静态变量。这意味着，Object Pascal 无法像 C++那样在应用程序的数据段中为类分配内存，而只能在堆中分配内存。

2.2.4 多态性

在 Object Pascal 语言中定义的类的方法通常是“静态”的，也就是在编译和链接阶段就确定了对象方法的调用地址。

面向对象的程序设计语言还可以在运行时才确定对象方法的调用地址，这种调用函数的方式叫做多态性，有时也称为动态联编或滞后联编。在 Object Pascal 语言中，多态性是通过虚拟方法或动态方法实现的。

通常，可以将类中的方法定义为下面的三种方式：

- 静态方法
- 虚拟方法
- 动态方法

在默认情况下定义的方法为静态方法，静态方法的调用地址在编译和链接的过程中就确定了。

在类中，如果定义了一个方法，在它的派生类中也可以定义一个同样的方法。对于静态方法，通常叫做“静态重载”。

下面的例子说明了静态方法 Infor 的调用情况。

```
program Project1;
{$APPTYPE CONSOLE}

type
TPerson = class // 人类
      procedure Infor; // 显示信息
end;
TEmployee = class(TPerson) // 职员类
      procedure Infor; // 显示职员信息
end;
```

```
procedure TPerson.Infor; // 显示调用的是TPerson类的Infor
begin
    Writeln ( 'TPerson.Infor');
end;
procedure TEmployee.Infor; // 显示调用的是TEmployee类的Infor
begin
    Writeln('TEmployee.Infor');
end;
var
    P1: TPerson; // 声明一个人类的变量
    E1: TEmployee; // 声明一个职员类的变量
begin
    P1:=TPerson.Create ;
    P1.Infor; // 调用的是TPerson类的Infor
    P1.Destroy;
    P1:=TEmployee.Create;
    P1.Infor; // 调用的是TPerson类的Infor
    TEmployee(P1).Infor;// 调用的是TEmployee的Infor
    P1.Destroy;
    E1:=TEmployee.Create;
    E1.Infor; // 调用的是TEmployee类的Infor
    E1.Destroy;
end.
```

运行结果如下：

```
TPerson.Infor
TPerson.Infor
TEmployee.Infor
TEmployee.Infor
```

可以看到，在“静态重载”的情况下， Infor 的调用是根据对象的类型来确定的。虚拟方法和动态方法也可以在派生类中被重载，通常称为“动态重载”。对象方法具体使用的方法并不是变量声明时指定的类的类型。

指明一个方法为虚拟方法或动态方法要使用指令字 virtual 或 dynamic。虚拟方法和动态方法在语意上是等价的，它们的不同主要之处在于运行时方法的调用实现上。虚拟方法调用的速度比较快，而动态方法的代码数量比较少。

一般在静态方法无法实现的情况下，可以考虑虚拟方法和动态方法实现动态联编。通常虚拟方法在实现多态性方面效率比较高。如果一个基类派生了很多子类，只是偶尔对方法进行重载，那么使用动态方法比较合适。

下面的例子说明了虚拟方法的定义与使用。

```
program Project1;
{$APPTYPE CONSOLE}
```

```
type
TPerson = class //  人类
      procedure Infor;virtual;
end;
TEmployee = class(TPerson) //  职员类
      procedure Infor;override;
end ;
TCustomer = class(TPerson) //  顾客类
      procedure Infor;override;
end ;
procedure TPerson.Infor;
begin
      Writeln ( 'TPerson.Infor ' ) ;
end;
procedure TEmployee.Infor;
begin
      Writeln( 'TEmployee.Infor');
end;
procedure TCustomer.Infor;
begin
      Writeln( 'TCustomer.Infor');
end;
var
      P1: TPerson; //  声明一个人类的变量
      E1: TEmployee; //  声明一个职员类的变量
begin
      P1: = TPerson.Create;
      P1.Infor; //  调用的是TPerson类的Infor
      P1.Destroy;
      P1:=TEmployee.Create;
      P1.Infor; //  调用的是TEmployee类的Infor
      T Worker(P1).Infor; //  调用的是TEmployee类的Infor
      P1.Destroy;
      P1:=TCustomer.Create;
      P1.Infor; //  调用的是TCustomer类的Infor
      TPerson(P1).Infor; //  调用的是TCustomer类的Infor
      P1.Destroy;
      E1:=TEmployee.Create;
      E1.Infor; //  调用的是TEmployee类的Infor
      TCustomer(E1).Infor; //  调用的是TEmployee类的Infor
      E1.Destroy;
end.
```

运行结果如下：

```
TPerson.Infor
TEmployee.Infor
TEmployee.Infor
TCustomer. Infor
TCustomer.Infor
TEmployee.Infor
TEmployee.Infor
```

2.2.5 类运算符

在程序运行期间，可以使用 is 运算符和 as 运算符来进行类信息检测和类型转换，通常也把这两个运算符称为运行时类型信息（RT T I：runtime type information）运算符。

1. is 运算符

is 运算符用来检测一个对象在运行时的类的类型，具体形式如下：

```
object is class
```

如果返回值为 True，那么对象 object 是类 class 或者是类 class 的派生类的一个实例。如果对象为 nil，返回值则为 False。

2 as 运算符

as 运算符用来进行类型转换检测的，具体形式如下：

```
object as class
```

返回值为 object 的一个引用，类型为 class 类型。在运行期间，object 必须是与 class 类兼容的一个类的对象或者为 nil。通常为了避免类型不兼容，可以使用 is 运算符来进行类型判断。

下面的例子对 is 运算符和 as 运算符进行了说明。

```
program Project1;
{$APPTYPE CONSOLE}
type
TPerson = class // 人类
public
    Name:string; // 姓名
end;
TEmployee = class(TPerson) // 顾客类
public
    DeptName:string; // 地址名称
    procedure Infor; // 只有子类具有"显示信息"的方法
end;
procedure TCustomer.Infor;
begin
```

```
    Writeln('姓名：',Name,';  部门名称：' ,AddressName);
end;
var
    P1: TPerson; //  声明一个人类的变量
    E1: TEmployee; //  声明一个顾客类的变量
begin
    P1:=TPerson.Create; // P1为父类的对象
    P1.Name:='张鹏';
    if P1 is TEmployee then (P1 as TEmployee).DeptName:='人事部' ;
    if P1 is TEmployee then (P1 as TEmployee).Infor;
    P1.Free;
    P1:=TCustomer.Create; // P1为子类的对象
    P1.Name:='冯斌';
    if P1 is TEmployee then (P1 as TEmployee).DeptName:='人事部' ;
    if P1 is TEmployee then (P1 as TEmployee).Infor;
    P1.Free;
    E1:=TEmployee.Create; // E1为子类的对象
    E1.Name:='高威';
    if E1 is TEmployee then (E1 as TEmployee).DeptName:='公关部';
    if E1 is TEmployee then (E1 as TEmployee).Infor;
    E1.Free;
    Readln;
end.
```

运行结果如下：

```
姓名：冯斌;  部门名称：人事部
姓名：高威;  部门名称：公关部
```

这里需要说明几点：

- 当 P1 调用 TPerson 类的构造函数的时候，创建的是一个 TPerson 的对象，不可以调用子类 TCustomer 中特有的过程 Infor，也不可以对子类中的特有属性 DeptName 进行操作。为了判断 P1 是否为 TCustomer 类的对象，使用了 is 运算符来进行判断。
- TPerson 类的变量 P1 被“创建”了两次，由于前后两次创建的类型不同，所以 is 运算符判断的结果不同。第 2 次“ P1 is TCustomer ”返回的是 True，然后进行类型转换并访问 DeptName 属性和调用 Infor 过程。

2.2.6 类方法和类引用

1. 类方法

一般的，类中所定义的方法被对象调用。在调用构造函数 Create 的时候，我们使用的是类，而不是具体的对象。类似地还可以定义一些类方法，它们对类进行操作，而不是对具体的对象进行操作。在定义类方法的时候，使用保留字 class 对过程或函数进行说明。下面的例

子说明了类方法的用法。

```
program Project1;
{$APPTYPE CONSOLE}

type
TEmployee = class // 职员类
    Name: string; // 职员姓名
    class procedure AddOne; //增加一个职员
    destructor Destroy;override; //析构函数，减少一个职员
end;

var
    ENum :integer; // 表示当前的职员数
    E1,E2: TEmployee; // 声明职员类的变量

{ TEmployee }
class procedure TEmployee.AddOne;
begin
    ENum:=ENum+1;
end;

destructor TEmployee.Destroy;
begin
    ENum:=ENum-1;
    inherited Destroy;
end;

begin
    E1: = TEmployee.Create;
    E1.AddOne; // 调用类方法改变变量ENum
    E1.Name:='李斌';
    Writeln( '职员数为：' ,ENum);
    E2:=TEmployee.Create;
    E2.AddOne;
    E2.Name:= '张华';
    Writeln('职员数为：',ENum);
    E1.Free;
    E2.Free;
    Writeln('职员数为：',ENum);
    Readln;
end.
```

运行结果如下：

```
职员数为：1
职员数为：2
```

职员数为：0

在定义类方法的时候，标识符 Self 将代表类方法被调用的类。不可以使用 Self 访问类的字段、属性和普通方法，但是可以通过 Self 调用构造函数和其它类方法。

2. 类引用

类引用（class reference）是一种数据类型，有时又称为元类(metaclass)，是类的类型的引用。

类引用的定义形式如下：

```
class of type
```

例如：

```
type SomeClass = class of TObject;
var AnyObj: SomeClass;
```

下面的例子说明了类引用的用法。

```
program Project1;
{$APPTYPE CONSOLE}

type
    TPerson = class //  人员类
    Name: string; //  姓名
end;
    TEmployee = class(THuman) //  职员类
    DeptName: string; //  部门名称
    procedure Infor; //  显示职员信息
end;
CRef = class of TObject; //  定义了一个"类引用"数据类型

var
    Employee: array[0..1] of TObject; //  类的变量数组
    i: Integer; //  循环变量
    CR:array[0..1] of CRef; //  类引用数组
    {TEmployee }
procedure TEmployee.ShowInf;
begin
    Writeln('姓名：',Name,';部门名称：' ,DeptName);
end;
begin
    CR[0]:=THuman; //  给类引用赋值
    CR[1]:=TEmployee;
for i:=0 to 1 do
begin
    Employee[i]:=CR[i].Create; //  创建对象
```

```
    if Employee [i] is TEmployee then // 判断对象的类型
begin
    (Employee[I] as TEmployee).Name:='金丰' ;
    (Employee[I] as TEmployee).DeptName:='人事部' ;
    (Employee[I] as TEmployee). Infor;
  end;
end;
  Readln;
end.
```

运行结果如下：

```
姓名：金丰  部门名称：人事部
```

注意：

上面定义了一个类引用类型的数组，其中的两个元素的数值分别为不同的两个类的类型。

2.2.7 单元文件

单元(unit)是组成 Pascal 程序的单独的源代码模块，单元由函数和过程组成，这些函数和过程能被主程序调用。

一个标准的单元文件格式如下：

```
unit Unit1；                //单元头

interface
uses ｛ 单元列表 ｝

{接口部分}

implementation
uses {单元列表}

{实现部分 }

initialization
{初始化部分}

finalization
{结束部分}

end
```

一个单元至少要由以下三部分组成：.

- 一个 unit 语句，每一个单元都必须在开头有这样一条语句，以标识单元的名称，单元的名称必须和文件名相匹配。例如，如果有一个文件名为 A1，则 unit 语句可能是：

```
unit A1
```

- interface 部分，在 unit 语句后的源代码必须是 interface 语句。在这条语句和

implementation 语句之间是能被程序和其它单元所共享的信息。一个单元的 interface 部分是声明类型、常量、变量、过程和函数的地方，这些都能被主程序和其它单元调用。这里只能有声明，而不能有过程体和函数体。interface 语句应当只有一个单词且在一行：

```
interface
```

- implementation 部分，它在 interface 部分的后面。虽然单元的 implementation 包含了过程和函数的源代码，但它同时也允许在此声明不被其它单元所调用的任何数据类型、常量和变量。implementation 是定义在 interface 中声明的过程和函数的地方， implementation 语句只有一个单词并且在一行上：

```
implementation
```

一个单元能可选地包含其它两个部分：

- initialization 部分，在单元中它放在文件结尾前，它包含了用来初始化单元的代码，它在主程序运行前运行并只运行一次。
- finalization 部分，在单元中它放在 initialization 和 end 之间。

注意如果几个单元都有 initialization/finalization 部分，则它们的执行顺序与单元在主程序的 users 子句中的出现顺序一致。不要使 initialization/finalization 部分的代码依赖于它们的执行顺序，因为这样的话主程序的 uses 子句只要有小小的修改，就会导致程序无法通过编译。

1．uses 子句

users 子句在一个程序或单元中用来列出想要包含进来的单元。例如，如果有一个程序名为 FooProg，它要用到在两个单元 UnitA 和 UnitB 中的函数和类型，正确的 uses 声明应该这样：

```
Program FooProg;

uses UnitA,UnitB;
```

单元能有两个 uses 子句，一个在 interface 部分，一个在 implementation 部分。这里有一个例子：

```
Unit FooBar

Interface

Uses BarFoo;
{在这里进行全局声明 }

implementation

uses BarFly;
{在这里进行局部声明 }

initiallization
{在这里进行单元初始化 }

finalization
```

```
{在这里进行推出操作 }
end.
```

2. 循环单元引用

读者也许经常会碰到这样的情况，在 UnitA 中调用 UnitB 并在 UnitB 中调用 UnitA，这称为循环单元引用。循环单元引用的出现表明了程序设计有缺陷，应该在程序中避免使用循环单元引用。比较好的解决方法是把 UnitA 和 UnitB 共有的代码移到第三个单元中。然而，有时确实需要用到循环单元引用，这时就必须把一个 uses 子句移到 implementation 部分，而把另一个留在 interface 部分，这样就能解决问题。

2.2.8 TObject：所有对象的祖先

因为所有对象都是从 TObject 继承来的，每一个类都从 TObejct 继承了一些方法，所以可以对对象的性能进行一些特殊的假定。每一个类都能告诉你它的名字、类型和它是否从某个类派生而来。作为一个程序员，不必关心编译器的实现细节而只要能利用对象所提供的功能就够了。

TObject 是一个特殊的对象，它在 system 单元中定义，编译器对 TObject 是完全清楚的，下面是 TObject 的定义：

```
type
  TObject=class
constructor Create;
procedure Free;
class function InitInstance(Instance:pointer):TObject;
procedure CleanupInstance;
function ClassType:TClass;
class function ClassName:ShortString;
class function ClassNameIs(const Name:String):Boolean;
class function ClassParent:TClass;
class function ClassInfo:pointer;
class function InstanceSize:longint;
class function InheritsForm(AClass:TClass):Boolean;
class function MethodAddress(const Name:ShortString):pointer;
class function MethodName(address:pointer):ShortString;
function FieldAddress(const Name:ShortString):pointer;
function GetInterface(const IID:TGUID;out obj):Boolean;
class function GetInterfaceEntry(const IID:TGUID):PinterfaceEntry;
class function GetInterfaceTable:PinterfaceTable;
function SafeCallException(ExceptObject:TObject;
  ExceptAddr:pointer):Hresult;virtual;
procedure AfterConstruction;virtual;
procedure BeforeDestruction;virtual;
procedure Dispatch(var Message);virtual;
```

```
procedure DefaultHandler(var Message);virtual;
class function NewInstance:TObject;virtual;
procedure FreeInstance;virtual;
destructor Destroy;virtual;
end;
```

在 Delphi 的联机帮助中可以看到每一个方法的文档。

在这里特别要注意那些前面有 class 关键字的方法。在一个方法前加上关键字 class，使得方法向其它通常的过程和函数一样调用而不需要生成一个包含这个方法的类的实例，这个功能是从 C++的 static 函数借鉴来的。要小心，不要让一个类方法依赖于任何实例信息，否则编译时将出错。

2.3 结构化异常处理

结构化异常处理（SHE）是一种处理错误的手段，使得应用程序能够从致命的错误中很好地恢复。在早期的 Delphi 中，异常是由 Object Pascal 语言来处理的；从 Delphi 2.0 开始，异常成为 Win32 API 的一部分。用 Object Pascal 来处理异常比较简单了，因为异常中包含了错误的位置和特征信息。这使得异常的使用和实现与普通的类一样。当一个错误或其它一些事件中止了程序的正常运行，系统就会抛出一个异常。

Delphi 中包含了一些预定义的通用的程序错误异常，例如内存不足、被零除、数字上溢和下溢以及文件的输入输出错误，你可以定义自己的异常类来适应程序的需要。通过 Delphi 的异常处理机制，可以捕获这个异常并进行处理。

异常实际上是一些对象，可以是任何类的一个实例。但是通常是我们自己定义的一个从 Exception 类派生出的异常类，定义方法与普通类的定义方法基本一致。Exception 类是在 SysUtils 单元中定义的。如果一个程序的 uses 语句中包含了 SysUtils 单元，发生运行错误时就会抛出一个异常。

可以利用类的继承性将一组异常组合成一个系列。比如，在 SysUtils 单元中就定义了有关数学方面的一组异常类：

```
type
EMathError = class(Exception);
EInvalidOp = class(EMathError);
EZeroDivide = class(EMathError);
EOverflow = class(EMathError);
EUnderflow = class(EMathError);
```

有时在异常类中还定义一些字段、属性和方法，通过它们可以传达一些错误信息。例如：

```
type
EInOutError = class(Exception)
ErrorCode: Integer;
end;
```

2.3.1 try...except 语句和 try...finally 语句

在 try...except 语句中可以进行抛出异常和处理异常的工作。try...except 的一般形式如下：

```
try
    Statements1；
except
    on Exception1 do HandleStatements1；
    on Exception2 do HandleStatements2；
    ……
on ExceptionN do HandleStatementsN；
else
    Statements2；
end.
```

对于有些操作，在异常处理部分要进行，在正常情况下也要进行。例如，在正常情况下，使用完文件之后关闭文件；如果在对文件操作的过程中出现了异常，也需要关闭已经打开的文件。这时，就可以把关闭文件的过程放在 try...finally 语句的 finally 部分，不管 try 部分的操作是否正常，都要进行 finally 部分的操作。

通常 try...finally 语句的形式如下：

```
try
    statementList1
finally
    statementList2
end
```

可以看到，try...finally 语句的用法与 try...except 语句的用法很相似。statementList1 可以为简单语句，也可以为复合语句。如果在 statementList1 中抛出了异常，程序立即转到 finally 部分；如果在 statementList1 中执行了 Exit、Break 或 Continue 过程而导致程序的控制离开 statementList1 部分时，程序也会跳转到 finally 部分；如果在 try 部分正常执行完毕，接着执行的还是 finally 部分。

下面的例子介绍了在文件输入/输出时，怎样用异常处理。可以区分与 try...except 语句和 try...finally 语句的用法。

```
Program Project1;
uses Classes,Dialogs;
{$APPTYPE CONSOLE}
var
  F:TextFile;
  S:String;
begin
  AssignFile(F, 'F1.TXT');
  try
reset(F);
```

```
try
   readln(F,s);
finally
   CloseFile(F);
end;
   except
on EinOutError do
   ShowMessage('Error in FileIO');
   end;
end.
```

在内层的 try...finally 代码块用来确保文件总是关闭的，而不管是不是发生了异常。这段代码的执行过程是：先执行 try 与 finally 之间的代码；如果执行完毕或出现异常，就执行 finally 与 end 之间的代码；如果确实有异常发生，就跳到外层的异常处理块。这样，即使出现异常，文件也总是关闭的，并且异常总能得到处理。

注意在 try...finally 块中，finally 后面的语句不管有没有异常都被执行。因此，finally 后面的语句不能以发生异常为前提。另外，由于 finally 后面的语句并没有处理异常，因此，异常被传递到下一层的异常处理块。

外层的 try...except 块用于处理程序中发生的异常。在 finally 中关闭文件后，except 块显示一个信息，告诉用户发生了 I/O 错误。

这种异常处理机制比传统的错误处理方式优越，它使得错误检测代码从错误纠正代码中分离出来。这是一件好事情，它会使程序更可读，它使得你能集中处理程序的其它代码部分。

使用 try...finally 代码块但不捕捉特定种类的异常是有一定意义的。当代码中使用 try...finally 块的时候，意味着程序并不关心是否发生异常，而只是想最终总是能进行某项任务。finally 块最适合于释放先前分配的资源(例如文件或 Windows 资源)，因为它总是执行的，即使发生了错误。不过，很多情况下，可能需要对特定的异常做特定的处理，这时候就要用 try...except 块来捕捉特定的异常，下面就是例子：

```
Program Project1;
{$APPTYPE CONSOLE}

var
   R1，R2:double;

begin
   while true do begin
   try
write('Enter a real number: ');
readln(R1);
write('Enter another real number: ');
readln(R2);
writeln('The first number divided by the Second is: ',(R1/R2):5:3);

 except
```

```
  on EInOutError do
    ShowMessage('It is not a valid number! ');
  on EZeroDivide do
    ShowMessage('Can not divide by zero! ');
    end;
    end;
end.
```

尽管在 try...except 块中可以捕捉特定的异常，也可以用 try...except...else 结构来捕捉其它异常。当使用 try...except...else 结构的时候，应当明白 else 部分会捕捉所有的异常，包括那些你并没有预料到的异常，例如内存不足或其它运行期异常。因此，使用 else 部分要小心，能不用则不用。当进入不合格的异常处理过程中时，你应当一直重触发这个异常。

2.3.2 raise 语句

使用 raise 语句调用一个异常类的构造函数，并抛出一个异常。例如：

```
raise EInOutError.Create;
```

通常，raise 语句的形式如下：

```
raise object at address
```

其中 object 和 at address 是可选项，address 通常是一个指向过程或函数的指针。一个抛出的异常在处理过后自动地被删除，一般不去主动地删除一个异常对象。

2.3.3 异常类

异常是一种特殊的对象实例，它在异常发生时才实例化，在异常被处理后自动删除。异常对象的基类被称为 Exception。

在异常对象中最重要的元素是 Message 属性，它是一个字符串，它提供了对异常的解释，由 Message 所提供的信息是根据产生的异常来决定的。

注意如果定义自己的异常对象，一定是要从一个已知的异常对象例如 Exception 或它的派生类派生出来的，因为这样通用的异常处理过程才能捕捉这个异常。

当你在 except 块中处理一个特定的异常时，可能会捕捉到该异常的派生异常。例如，EMathError 是所有与数学有关的异常(例如 EZeroDivide、EOverflow)的祖先。

凡是没有显式地处理的异常最终将被传送到 Delphi 运行期库中的默认处理过程并在此得到处理。

2.4 方法与技巧

2.4.1 设置代码模板

代码模板是 Delphi 的代码感知特性的一种，通过它可以快速、高效和正确地输入代码。代码模板将一些常用的语句块保存在模板中，然后程序员只要在代码编辑器中按下 Ctrl+J，则会弹出一个小窗口，其中对已有的语句块进行了列表显示。可以选择其中的一个并按下回

车键，则该模块语句就整个地出现在当前光标所在的位置。

例如，在需要循环语句的地方，可以选中代码模板列表窗口中的“ for statement”，当前光标所在位置就会出现下面的代码：

```
for := to do
begin
end;
```

接着只要将这些代码完善就可以了。

下面介绍如何添加、删除代码模板。可以在代码模板中添加自定义的模板，具体过程如下：

（1）通过菜单 Tools/Editor Options...打开 Editor Properties 对话框，选中其中的 Code Insight 标签。

（2）按下 Add...按钮，在弹出的 Add Code Template 对话框中输入代码模板的名称和描述，完成后按下 OK 按钮，返回到 Code Insight 标签中。

（3）现在 Templates 项中选中的条目就是刚才新加的一个代码模板。在 Code 项中输入代码模板中包含的代码。

（4）按下 OK 按钮后就可以在代码编辑器中使用新定义的代码模板了。

（5）如果要删除已有的代码模板，只要在 Code Insight 标签的 Templates 项中选中要删除的代码模板，然后按下 Delete 按钮就可以了。

（6）如果按下 Code Insight 标签中的 Edit...按钮，可以修改 Templates 项中当前选中代码模板的名称和描述。

2.4.2 设置提示信息

在程序开发的过程中，可能会声明一个或多个变量，但是这些变量却始终没有被使用。在默认的情况下，编译器会提示这些变量在声明后从来没有被使用过。通过下面的方法，可以决定这些消息是否显示。

- 通过改变关于工程的一些环境设置来实现。可通过菜单命令 Project/Options...打开 Project Options 对话框，选中 Compiler 标签，改变 Messages/ Show hints 选项即可。
- 通过编译指令{$HINTS ON}与{$HINTS OFF}来进行设置。下面这段代码就是在上面 Messages/Show hints 选项被选中的情况下实现不显示此类提示信息的。

```
...
{$HINTS OFF}
procedure Test;
var
I: Integer;
begin
end;
...
```

在编译上面这段代码的过程中，不会出现变量 I 没有被使用的提示信息。

2.5 本章小结

本章的内容相当丰富，主要介绍了 Object Pascal 语言的基本语法和语义，其中包括：变量、运算符、数据类型、函数、过程和类型；同时对面向对象编程、对象、域、属性、方法、TObject、接口、异常处理有了了解。现在，读者应该已经理解了面向对象的 Object Pascal 的工作原理，下面可以进行可视化组件库（VCL）学习了。